Advancing Traffic Efficiency and Safety through Software Technology phase 2 (ATESST2)

Report type: D2.1 Appendix A3.2
Report name: Review of relevant Safety Analysis Techniques

Dissemination level: PU
Status: Final
Version number: 1.1
The Consortium
Volvo Technology Corporation (Sweden)
VW/Carmeq (Germany)
Centro Ricerche Fiat (Italy)
Continental Automotive (Germany)
Delphi/Mecel (Sweden)
Mentor Graphics Hungary (Hungary)
CEA LIST (France)
Kungliga Tekniska Högskolan (Sweden)
Technische Universität Berlin (Germany)
University of Hull (UK)
### Revision chart and history log

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>20/03/09</td>
<td>Integration of UOH &amp; CRF sections.</td>
</tr>
<tr>
<td>0.2</td>
<td>23/03/09</td>
<td>Further editing and fixing of problems as per suggestions by KTH_MT</td>
</tr>
<tr>
<td>0.3</td>
<td>27/03/09</td>
<td>Final editing, additions of VTEC &amp; further CRF input</td>
</tr>
<tr>
<td>1.0</td>
<td>27/03/09</td>
<td>Released version</td>
</tr>
<tr>
<td>1.1</td>
<td>1/06/10</td>
<td>Final adaptations for use as appendix in D2.1</td>
</tr>
</tbody>
</table>
List of abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AADL</td>
<td>Architecture Analysis &amp; Design Language</td>
</tr>
<tr>
<td>CART</td>
<td>Classification and Regression Trees</td>
</tr>
<tr>
<td>CCD</td>
<td>Cause Consequence Diagrams</td>
</tr>
<tr>
<td>CEG</td>
<td>Cause Effect Graphs</td>
</tr>
<tr>
<td>CFT</td>
<td>Component Fault Trees</td>
</tr>
<tr>
<td>DBR</td>
<td>Dragonian Blood Rituals</td>
</tr>
<tr>
<td>DCCA</td>
<td>Deductive Cause Consequence Analysis</td>
</tr>
<tr>
<td>DDT</td>
<td>Diagnostic Decision Trees</td>
</tr>
<tr>
<td>DFOA</td>
<td>Deductive Failure Order Analysis</td>
</tr>
<tr>
<td>DSPN</td>
<td>Deterministic Stochastic Petri Nets</td>
</tr>
<tr>
<td>FADDe</td>
<td>Formal Analysis using Dangerous and Deviant Leprechauns</td>
</tr>
<tr>
<td>FMEA</td>
<td>Failure Modes &amp; Effects Analysis</td>
</tr>
<tr>
<td>FMECA</td>
<td>Failure Modes Effects &amp; Criticality Analysis</td>
</tr>
<tr>
<td>FPTC</td>
<td>Failure Propagation &amp; Transformation Calculus</td>
</tr>
<tr>
<td>FPTN</td>
<td>Failure Propagation &amp; Transformation Notation</td>
</tr>
<tr>
<td>FSAP</td>
<td>Formal Safety Analysis Platform</td>
</tr>
<tr>
<td>FTA</td>
<td>Fault Tree Analysis</td>
</tr>
<tr>
<td>GDE</td>
<td>General Diagnostics Engine</td>
</tr>
<tr>
<td>GTST</td>
<td>Goal Tree Success Tree</td>
</tr>
<tr>
<td>HiP-HOPS</td>
<td>Hierarchically Performed Hazard Origin and Propagation Studies</td>
</tr>
<tr>
<td>MBSA</td>
<td>Model-based safety analysis</td>
</tr>
<tr>
<td>MFM</td>
<td>Multilevel Flow Model</td>
</tr>
<tr>
<td>RTES</td>
<td>Real-Time Embedded Systems</td>
</tr>
<tr>
<td>SDA</td>
<td>Software Deviation Analysis</td>
</tr>
<tr>
<td>SEFT</td>
<td>State-Event Fault Trees</td>
</tr>
<tr>
<td>UDDER</td>
<td>Unicorn-based Division of Deviations, Errors, and Reliability</td>
</tr>
</tbody>
</table>
# Table of contents

Authors ................................................................. 2
Revision chart and history log.................................. 3
List of abbreviations........................................... 4
Table of contents ............................................... 5
List of Figures ................................................... 7

1 Introduction ................................................... 7

2 Safety Analysis of System Designs ......................... 9

2.1 Compositional approaches ................................ 10
   2.1.1 Failure Propagation & Transformation Notation 10
   2.1.2 Failure Propagation & Transformation Calculus 11
   2.1.3 Component Fault Trees ................................ 12
   2.1.4 State-Event Fault Trees ............................. 14
   2.1.5 HiP-HOPS............................................. 15
   2.1.6 AADL................................................ 17

2.2 Behavioural Fault Simulation approaches ............ 22
   2.2.1 Altarica............................................ 23
   2.2.2 FSAP/NuSMV ...................................... 24
   2.2.3 Model-based Deviation Analysis................... 27
   2.2.4 DCCA............................................. 30
   2.2.5 Automatic FMEA Approaches..................... 31

3 Online Fault Monitoring & Diagnosis .................... 32

3.1 Introduction .................................................. 34
3.2 Rule-based expert systems ............................. 34
3.3 Model-based approaches ................................ 36
   3.3.1 Fault propagation models ....................... 36
   3.3.2 Causal Process graphs .......................... 38
   3.3.3 Functional decomposition ....................... 40
   3.3.4 Behavioural models ................................ 42
   3.3.5 Qualitative simulation.......................... 44
3.4 Data-driven approaches ................................. 45

4 Safety design flow............................................. 48

4.1 System functional analysis ............................. 49
4.2 System-Vehicle interaction analysis .................. 51
4.3 System boundary definition ........................... 53
4.4 Scenarios definition ..................................... 56
4.5 Hazard identification .................................................................................................. 80
  4.5.1 Malfunctions definition ..................................................................................... 80
  4.5.2 Misuses and maintenance condition definition ................................................ 81
  4.5.3 Hazard definition .............................................................................................. 81
4.6 Risk analysis .......................................................................................................... 82
  4.6.1 Scenarios tree definition ................................................................................... 82
  4.6.2 Controllability level estimate ............................................................................ 82
  4.6.3 Severity estimate ............................................................................................. 83
  4.6.4 Scenarios tree reduction .................................................................................. 84
  4.6.5 Exposure time estimate .................................................................................... 84
  4.6.6 ASIL determination ......................................................................................... 84
4.7 Safety Goals and Safe States definition ................................................................. 85
4.8 Risk assessment approval ...................................................................................... 87
  4.8.1 Cooper-Harper Scale ...................................................................................... 87
4.9 Functional safety requirements definition .............................................................. 88
  4.9.1 System FMEA vs Risk assessment ................................................................... 89
4.10 Technical safety requirements definition .............................................................. 91
4.11 ASIL allocation ..................................................................................................... 93
  4.11.1 ASIL Decomposition .................................................................................... 93
  4.11.2 Criticality Analysis ....................................................................................... 101
4.12 Current EAST-ADL language support for safety workflow .................................. 102
  4.12.1 Identification of Safety-related information ................................................... 102
5 Conclusions ................................................................................................................ 106
  5.1 Safety Analysis of System Designs ...................................................................... 106
  5.2 Online Fault Monitoring & Diagnosis ................................................................. 107
  5.3 Safety Design Flow ............................................................................................. 108
  5.4 Final Summary .................................................................................................... 109
6 References ................................................................................................................ 110
List of Figures

Figure 1 - Example FPTN component ........................................................................12
Figure 2 - Example RTN graph ...............................................................................13
Figure 3 - Component Fault Tree .............................................................................15
Figure 4 - SEFT notation (from [13]) ......................................................................16
Figure 5 - The synthesis of fault trees from the system model ..............................18
Figure 6 - Failure annotations of a computer-operated two-way valve ...............20
Figure 7 - Example system and fragments of local safety analyses ....................21
Figure 8 - Mechanically constructed fault tree for the example system ..........22
Figure 9 - The conversion of fault trees to FMEA .................................................23
Figure 10 - Simple triple redundant system ...........................................................25
Figure 11 – Disjunctive Completion Law ...............................................................27
Figure 12 - AADL Graphical Notation of Components [22] .................................33
Figure 13 - Connections and Port Graphical Representations in AADL ............34
Figure 14 – Abstractions of AADL Elements [22] ................................................35
Figure 15 – Error Model Definition for a Component without propagation ........36
Figure 16 - Error Model Definition for a Component with in out propagation ....37
Figure 17 - Error Propagations between Error Model Instances [25] ...............42
Figure 18 - Guard_In Property for Rules that handle Incoming Propagations [25]43
Figure 19 - Graphical Representation of an AADL Sample Specification ............46
Figure 20 - Subcomponents with Data and Event Ports ......................................48
Figure 21 - Cause Consequence Diagram .............................................................65
Figure 22 - Fault Trees as part of a dynamic behavioural model ........................66
Figure 23 - An example Digraph .........................................................................67
Figure 24 - Goal Tree Success Tree (GTST) ..........................................................69
Figure 25 - Main phases of the safety lifecycle .....................................................73
Figure 26 – Malfunction\Scenario\Hazard relation ...............................................81
Figure 27 - Risk Definition ....................................................................................85
Figure 28 – Cooper-Harper scale ......................................................................87
Figure 29 - Cooper-Harper example ................................................................88
Figure 30 – System FMEA process ....................................................................89
Figure 31 – Component FMEA process ..............................................................91
Figure 32 - Technical safety requirements definition process .............................92
Figure 33 - Decomposition schemes ....................................................................94
Figure 34 - Block diagram for steering system ...................................................95
Figure 35 - Risk assessment abstract – Row related to the situation at maximum risk level ....97
Figure 36 - Safety Channel ...............................................................................98
Figure 37 - Data flow diagram ................................................................. 98
Figure 38 - ASIL decomposition – first step ............................................. 99
Figure 39 - ASIL partitioning schemes .................................................... 99
Figure 40 - ASIL decomposition ......................................................... 100
Figure 41 - Identification of safety-relevant information ..................... 103
Figure 42 - Requirements constructs related to safety ......................... 105
1 Introduction

This document contains a review of techniques for model-based safety analysis and fault diagnosis, including a discussion of the implications of emerging automotive safety standards. The ultimate goal of these techniques is to enable the design of more dependable systems, i.e. systems which have a greater probability of being able to carry out their desired functions even if exposed to undesirable conditions.

The first area to be reviewed is the field of model-based safety analysis. Model-based safety analysis (MBSA) is a valuable technique in which safety engineers perform their analyses using the design model of the system, produced as part of a model-based design process; it can bring a number of benefits to the design of safety-critical systems, particularly the design of real-time embedded systems (RTES), which are increasingly complex and which are frequently distributed across a networked or cooperative system architecture. RTES can improve the safety of a system, e.g. Electronic Stability Control in vehicles has been shown to be effective in maintaining control and saving lives by significantly reducing the number and severity of crashes [1]. RTES and similar systems are widely used in safety-critical industries such as the automotive and aerospace industries. Although there is an increasing trend towards the use of correct-by-construction techniques in embedded systems, it is nevertheless vital to be able to perform a thorough and accurate safety analysis of those systems to ensure they meet their dependability requirements. By identifying areas in a system where reliability or safety is deficient, actions can be taken to remedy the weaknesses and thereby improve the design of the system.

The second field to be examined is that of online fault diagnosis. Model-based safety analysis techniques help to create a safe and reliable design for a system, but once that system has been created and has entered operation, a new set of techniques are required for monitoring and diagnosing any faults which may occur. This is a particularly critical task as while the previous techniques deal only with potential faults, fault diagnosis involves detecting and diagnosing actual faults that can pose a very serious risk to both the system and its environment so that error handling techniques can solve or mitigate those faults. In such situations, the error handling will frequently require a fast and accurate diagnosis in order to solve or mitigate any problems that may arise, and this presents a different set of challenges compared to offline safety analysis. Fault diagnosis techniques must therefore be able to quickly establish the root cause of a system malfunction, often on the basis of limited monitoring information, if the worst consequences of the malfunction are to be averted.

Finally, we shall also take a look at the implications of the new automotive safety standard ISO26262, particularly with regard to verification and validation techniques. In designing a modelling language for automotive systems like EAST-ADL, it is important not only to study the state of the art in these areas but also to examine the state of the art in industrial practice as envisaged by new standards like ISO 26262.

On the basis of this review, we hope to be able to help clarify the way forward for the project.
2 Safety Analysis of System Designs

Traditional safety analysis has typically operated on an informal understanding of the system design. Such techniques include fault tree analysis (FTA) [UOH2,3], in which the combinations of possible causes are deduced from the system failure, and Failure Modes & Effects Analysis (FMEA) [4], which analyses the possible effects each failure can have on the system. FMECA (Failure Modes, Effects, & Criticality Analysis) is also commonly used and is an extension of FMEA which includes an assessment of the criticality of failures, analysing their probability and severity. These types of techniques are often primarily manual processes, carried out either by a single person or a team of engineers, in order to produce comprehensive documents to fulfil safety requirements and to devise strategies to mitigate the effects of failure [5]. Although a great deal of valuable knowledge about the safety and reliability of the system is gained in the process, this type of informal, ad-hoc approach has a number of drawbacks.

Firstly, because the analysis takes place using informal knowledge of the failure behaviour of the system, the safety analysis is stored separately from the knowledge of the structure of the system, which is typically modelled more formally, and this can result in discrepancies or inconsistencies. Secondly, the primarily manual nature of the analysis process increases the risk of introducing errors or producing an incomplete analysis, particularly as the systems in question grow more intricate. Furthermore, a manual analysis is usually much more difficult and expensive, meaning that it is rarely carried out more than once and often only at the end of the design process to ensure that the design meets safety requirements, despite the potential benefits that multiple safety analyses can yield when used as part of an iterative design process. Finally, the informal nature of the results of such ad-hoc analysis makes it difficult to reuse that information, whether in a future iteration of the same system or in the design of a new system, particularly because the safety information is stored mainly within the results of separate analyses and is therefore separated from the system design itself.

In model-based safety analysis (MBSA), by contrast, the safety analysts and the system designers use the same model of the system, or at least models which are closely linked in some way, and this has a number of important benefits. Firstly, the resulting model is often more formal than a separate safety analysis and this can introduce the possibility of automating part of the process of safety analysis, e.g. by automatically generating fault trees from the system model or by simulating the failure behaviour of the system by injecting faults into the model. This not only simplifies the process, it also saves time and more importantly enables the safety analysis to be used as part of an iterative design process because new results can more easily be generated once the model has been changed. The more structured nature of the modelling also reduces the probability of introducing errors or of omitting important detail since the safety information is linked to the structural/nominal model of the system. Finally, a model-based safety analysis is often much more suitable for reusing in other projects, because the safety information is packaged with the system model, usually at a component-level, making it possible to reuse parts of the system without necessarily needing to perform another separate safety analysis.

Consequently, over the last 15 years, research has focused on further simplifying safety assessment by automating the synthesis process. This work has followed two different paradigms, each defining a distinct way of synthesising system failure models from other system information. The first paradigm can be called compositional failure analysis while the second behavioural fault simulation [6, 7]. Compositional techniques are usually deductive in nature (going backwards from failure to determine the cause) while behavioural techniques are typically inductive in nature (going forwards from a failure to determine the effect). Some of the more prominent examples in each category will be discussed below.
2.1 Compositional approaches

In compositional failure analysis, system failure models are constructed from component failure models using a process of composition. System failure models typically comprise – or can be automatically converted into – well-known dependability evaluation models such as fault trees, stochastic Petri-nets and Markov chains. These types of techniques therefore model the abnormal rather than normal behaviour of the system; the use of a separate model (or separate annotations of a model) makes it easier to analyse the effect of failures on the system but means that additional effort is required to create this new model or extend any normal system model with the required information, and further effort may be required to harmonise these disparate models. The true benefits of this type of approach are most apparent when used as part of an iterative design process; because the failure behaviour of the system components is modelled in a compositional fashion, it is much easier to determine the effects that changing one component or subsystem will have on the rest of the system. This is particularly true of automated or partly automated techniques, which speed up the analysis process and make it possible to rapidly evaluate speculative changes to the design.

Techniques that follow the compositional approach include: Failure Propagation and Transformation Notation, HiP-HOPS, Component Fault Trees, State-Event Fault Trees and Failure Propagation and Transformation Calculus. The Error Model Annex of the AADL also falls into this category as components are hierarchically annotated with state-based failure information, which can then be subsequently analysed by tools [8]. These different techniques will be discussed next.

2.1.1 Failure Propagation & Transformation Notation

Failure Propagation & Transformation Notation or FPTN is a simple graphical technique designed to overcome the limitations of FTA and FMECA in representing the failure behaviour of systems [9]. It consists of a modular, hierarchical notation for describing the propagation of faults through the modules of a system architecture. In addition to basic propagation, failures can also be transformed from one type into another, e.g. a watchdog timer may malfunction, causing a timeout and forcing a process to return an approximate value instead, converting a timing failure into a value failure. Failures in FPTN are therefore typed according to different types of failure domains.

Each module in FPTN is represented as a box and possesses a set of inputs and outputs that connect to other modules in the system; each module can also contain a number of sub-modules to create a component hierarchy [10]. The relation between the inputs and outputs is expressed by a set of logical equations equivalent to the minimal cut sets (the smallest combination of failures required to cause a higher-level fault) of the fault trees describing the output failure modes of the module; in this way, each module represents a number of fault trees describing all the failure modes for that module. These equations can also contain more advanced constructs, allowing FPTN to represent recovery mechanisms and internal failure modes. This type of notation enables FPTN to be used both inductively, to create an FMECA (by tracing a single component failure through the system to determine its effects), and deductively, to create an FTA (by tracing a single output failure to its root causes). In this way it acts to bridge these two techniques.
Figure 1 shows a simple component in FPTN named Subsystem1. The EH to the top-left indicates it contains an error handler and the II* to the top-right indicates the criticality class of the component. The arrows on the left are inputs and the arrows on the right are outputs, each annotated with a name and type (e.g. A:t means input A of type 't', where 't' is an abbreviation for 'timing'; similarly, 'v' for 'value' and 'o' for 'omission'). The FPTN expressions inside the box indicate the relation between inputs and outputs, e.g. in this case, X:t is caused by a combination of A:t and B:v. C:t has no effect as it is handled by the error handler.

FPTN is designed to be developed alongside the design of the system, so that as the system model evolves, so does the failure model. Information gleaned from an analysis of the FPTN model can then be used to identify potential flaws and problems in the system design so that they can be eliminated or compensated for in the next design iteration. However, although FPTN provides a systematic and formal notation for representing the failure behaviour of a system (a distinct improvement on earlier ad hoc approaches), it lacks full automation, meaning that its ability to be used in an iterative design process is hindered by the fact that each analysis must be conducted manually.

2.1.2 Failure Propagation & Transformation Calculus

Failure Propagation & Transformation Calculus (FPTC), not to be confused with FPTN above, is a method for the modular representation and analysis of the failure behaviour of a system's software and hardware components [11]. It allows an analyst to annotate an architectural model of a system with concise expressions describing how each component can fail; these annotations can then be used to compute the failure properties of the whole system automatically.

FPTC is designed primarily for the hard real-time software domain, and as such its primary unit of architectural description is the statically schedulable code unit, representing a single sequential thread of control. These units are then connected via communications protocols (handshakes, buffers etc) which capture the the data and control flow behaviour. FPTC assumes that the architecture is static, i.e. that all threads and communications are known in advance and are not dynamically created and destroyed during system operation. FPTC also offers the capability to describe the allocation of these units and their communications to different physical processing devices and networks. This makes it possible to describe how, for example, one corrupt processor can affect all software units running on it.

FPTC represents the system architecture using a RTN (Real-Time Network) style notation, consisting of a graph of arcs and nodes representing hardware and software units and the communications between them. Communications are typed according to protocol (e.g. blocking / non-blocking). RTN offers significant capabilities, including the ability to refine graphs hierarchically, to define code units as state machines, and to automatically generate Ada code.
from the design. An example RTN graph and associated key to some communications protocols are displayed below:

![Figure 2 - Example RTN graph](image)

<table>
<thead>
<tr>
<th>destructive read (blocking)</th>
<th>non-destructive read (non-blocking)</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-destructive write (blocking)</td>
<td>channel</td>
</tr>
<tr>
<td>destructive write (non-blocking)</td>
<td>signal</td>
</tr>
</tbody>
</table>

Each component is then considered to determine how it might respond to potential input failures. These failures are typed similarly to FPTN above, e.g. timing failures, value failures, sequence failures, but the types are not fixed and can be extended as required. 'Normal' is also a type, indicating the lack of a failure. Components may respond in one of two ways: by propagating the failure (so that a timing failure at the input will be passed on as a timing failure at the output) or by transforming the failure (e.g. converting a timing failure into a value failure). Components may also originate or terminate failures, e.g. by failing silent or by detecting and correcting failures.

The reaction to failure is described by a simple pattern-based notation; for example, if a component originates a failure, then it may be described as:

```
* → late
```

where * means normal input behaviour and late means an output failure of the 'late' type. Similarly, propagation would be:

```
omission → omission
```

etc. A component is typically annotated with a number of such expressions, known as clauses, to describe its overall behaviour in reaction to different types of failures. Both components and the communications between them can be annotated, since communication protocols can also introduce or transform failures.

The resulting RTN graph, once annotated with FPTC expressions, can then be thought of as a token-passing network in which failure tokens flow from one node to another, being created, transformed, and destroyed along the way. Each arc in the graph can then be further annotated with the set of possible failures that may propagate along it. This is done by 'running' each expression in reaction to the 'normal' type (the *) and listing the resulting output failures on the output communication arcs; each component is then re-run in response to any new input failure.
types. The process terminates when no more new output failures are generated and thus all possible input/output combinations have been considered.

FPTC can also handle combinations of inputs/outputs by means of tuples of failures, e.g.:

\[ \text{lute} \rightarrow (\text{value, late}) \]

here, a 'late' failure at the sole input will lead to a value and another late failure at the two outputs. It is also possible to use wildcards for inputs, e.g. \((\text{late, _})\) means a late failure in the first input and any type of failure (including 'normal') in the second input.

FPTC offers a significant advantage over FPTN: whereas FPTN is annotated according to known failures, and so any new failure to be added requires the whole model to be manually reannotated accordingly, FPTC uses the full architectural models used for developing the software code and can adapt much more readily to changes. This helps ensure the FPTC model is synchronised with the design. However, the primary disadvantage of FPTC is the necessity of performing a different analysis for each failure or combination of failures to be considered. Each originating failure must be specified at a component and then the model must be re-analysed to determine how this failure will propagate through the system and what failure modes will be communicated to critical components.

### 2.1.3 Component Fault Trees

Component Fault Trees [12] is an approach that aims to provide better support for hierarchical decomposition and overcome the disassociation between the hierarchy of faults in a normal fault tree and the architectural hierarchy of the system components. In the CFT approach, each component is represented by its own extended fault tree (the Component Fault Tree) which can also represent inputs and output ports as well as basic events and logical gates. This means that the CFTs can be integrated more closely into the system model. Because CFTs are still logical structures linking output failures to input causes, they can still be analysed (both qualitatively and quantitatively) using standard fault tree algorithms.

The rationale behind the CFT approach is that although hierarchical decomposition can be a useful tool to overcome the complexity of modern systems, creating a normal fault tree means examining all levels at once and thus cannot benefit from the decomposition. CFTs instead define smaller fault trees for each component, thus incorporating the fault trees as part of the hierarchical component model of the system. This means that CFTs for different parts of the system can be developed separately, or even stored as part of the component definition in a library of component types, facilitating a greater degree of reusability. The benefit is a flexible and efficient method of providing a description of the failure behaviour of the system as part of the model itself. Conceptually, this hierarchical decomposition also makes it possible for the failure behaviour of the system to be modelled at different levels, e.g. for the top level subsystems first, and then once the design has been refined further, for the sub components as well.

The fault trees used in the CFT approach are actually a variation on classical fault trees called 'Cause Effect Graphs' or CEGs. These differ from normal fault trees in allowing multiple top events and by representing repeating (or common cause) failures only once. As a result, CEGs are a type of directed acyclic graph rather than a true tree structure. The use of CEGs makes the CFTs smaller and easier to analyse, both significant benefits when modelling large systems. It also makes the diagrams clearer, as the fault tree nodes can be displayed as part of their components (see Figure 3).
Component Fault Trees serve as a very simple and effective way of embedded fault trees – and thus a description of the failure behaviour – inside a hierarchical, architectural model of the system. Because they are still normal fault trees in most respects, classical FTA algorithms still apply, and it is possible to determine the minimal cut sets for all system outputs – i.e. what combinations of component failures can cause the system failures – as well as the probability of those system failures by means of quantitative analysis.

2.1.4 State-Event Fault Trees

Fault Tree Analysis, while a popular technique, does have a number of faults of its own. One of these is its inability to adequately the temporal order of events, whether in terms of a simple sequence or a set of states and transitions. This drawback is particularly limiting when attempting to analyse complex systems, particularly real-time embedded and software based systems. Fault trees are fundamentally a combinatorial technique and are not well suited to modelling the dynamic behaviour in such systems.

State-Event Fault Trees or SEFTs [13] are designed to help overcome this limitation by adding capabilities for representing states and events to fault trees, thereby making it possible to use system state-based models as the basis for the analysis (instead of having to create new, static models) as well as enabling the use of more sophisticated analysis methods, e.g. Markov chains. States are taken to be conditions that persist over a period of time whereas events are instantaneous (and can trigger changes of state). SEFTs can also be seen as an evolution of CFTs above in that they allow decomposition and distribution across the components of the system, and represent inputs and outputs as visible ports in the model. Quantitative analysis of SEFTs is possible by means of converting them into Deterministic Stochastic Petri Nets (DSPNs) [14, 15]; these can then be analysed by separate external tools automatically, e.g. TimeNET [16].
SEFTs make a distinction between causal and sequential relations, the former applying to events and the latter to states, and therefore provides corresponding separate types of ports. Because events are explicitly represented (and do not always have to be state transitions), it is also possible for one event to cause another event. These events can also be combined using traditional fault tree gates (e.g. AND and OR) so that a combination of events is necessary to trigger another event. An example of the basic SEFT notation is given in Figure 4:

![Figure 4 - SEFT notation (from [13])](image)

Each component has its own state space and each component can only be in one state at any point in time (the ‘active state’). State expressions can be used to determine whether or not a given state is the active state, and such expressions can also make use of the logical gates; this means that operators such as AND and OR can also be applied to states, e.g. so that two or more separate states must be true at a given point in time for some event to occur. For the purposes of quantitative analysis, probabilities can be assigned to each state to reflect its chance of being the active state at any time. Similarly, events may be assigned probability densities for quantitative analysis.

SEFTs also offer more advanced features for modelling timing scenarios. For example, events can be assigned deterministic or probabilistic delays by means of Delay Gates and SEFTs also allow the use of NOT gates. Sequential and causal modelling is further refined by means of History-AND and Priority-AND gates, which can check whether an event has occurred in the past and in what
order it occurred, and other gates are also possible, e.g. Duration gates to ensure that a state has been active for a given amount of time.

Modelling of system failure behaviour using SEFTs follows the same general procedure as with standard FTA: namely, the analyst begins with the occurrence of a system failure and traces it back through the components of the system to determine its root causes. SEFTs differ in the level of detail possible during this analysis, e.g. in considering what states the components must be in to allow the failure to occur. SEFTs also allow a greater degree of reuse than traditional fault trees because pre-existing state charts from the design can be used, as can Markov chains, which can be similarly integrated into the SEFTs.

SEFTs differ from CFTs in that they can no longer be analysed using traditional FTA algorithms. The inclusion of states and the different modelling of events means that different techniques are needed, such as the conversion to Petri Nets, to allow for the calculation of probabilities of system failures. This requires the consideration of the entire system, which can lead to an explosion of state-spaces and thus performance problems for larger system models. This issue can be alleviated to some degree by using both combinatorial FTA-style algorithms and dynamic state-based algorithms to analyse different parts of the system, e.g. using the faster techniques for simple, static subsystems and using slower but more powerful techniques for the dynamic parts of the system. Clearly, however, the effectiveness of this dual-analysis technique will depend heavily on the type of system being analysed.

### 2.1.5 HiP-HOPS

Hierarchically Performed Hazard Origin & Propagation Studies, or "HiP-HOPS", is a comprehensive safety analysis methodology similar in purpose to previous approaches such as FPTN and CFTs but with more features and a greater degree of automation [17]. In this technique fault trees and FMEA are automatically constructed from topological models of the system that have been augmented with appropriate component failure data.

#### 2.1.5.1 Overview of the methodology

A HiP-HOPS study of a system under design has four phases:

- System modelling and failure annotation
- Fault Tree synthesis
- Fault Tree analysis & FMEA synthesis
- Optimisation

The first phase is executed manually while the later three phases are fully automated.

The first phase consists of developing a model of the system (hydraulic, electrical or electronic, mechanical systems, and conceptual block and data flow diagrams) and then annotating components in that model with failure data. In this phase, modelling can be carried out in a modelling tool like Matlab Simulink as it would normally have been carried out for the purposes of simulation. Failure annotations are then added to components of the model.

The second phase is the fault tree synthesis process. In this phase, an automated algorithm is applied to the annotated system model to create a set of fault trees which define system failures and their causes in the architecture. The algorithm works by taking failures of system outputs and progressively working backwards through the model to determine which components caused those failures. System failures and component failures are then joined together using the appropriate logical operators to construct fault trees with the failures at the system outputs as the top events and the root causes as basic events. The concept is illustrated in Figure 5.
In the third phase, fault trees are analysed and an FMEA is constructed which combines all the information stored in individual fault trees. The FMEA is presented in the form of a table that lists, for each component, the effects of each component failure on the system. As part of this process, both qualitative (logical) and quantitative (numerical-probabilistic) analyses are carried out on the fault trees. These analyses provide both the minimal cut sets of each fault tree and the unavailability (i.e. failure probability) of top events.

Figure 5 - The synthesis of fault trees from the system model

If the system is to be optimised, then the fourth phase is applied. This is an iterative optimisation of the system model, using genetic algorithms to continually evolve the model to find the best configuration of components on the basis of safety and cost. The optimisation phase will be described more fully later.

2.1.5.2 Modelling Phase

HiP-HOP studies can be performed on any model of a system that identifies components and the material, energy or data transactions among components. Such models can be hierarchically arranged, to manage complexity, if necessary. The basic idea of HiP-HOP is that an output failure of a component can either be caused by an input failure, an internal failure, or some combination of both. The local component output deviations and topology information are used to determine the relation between local deviations and top events.

For the purpose of the analysis, each component in the model must have its own local failure data, which describes how the component itself fails and how it responds to failures propagated by other components in the vicinity. Essentially, this information specifies the local effects that internally generated or propagated failures have on the component’s outputs. This is achieved by annotating the model with a set of failure expressions showing how deviations in the component outputs (output deviations) can be caused either by internal failures of that component or corresponding deviations in the component’s inputs. Such deviations include unexpected omission of output or unintended commission of output, or more subtle failures such as incorrect output values or the output being too early or late. This logical information explains all possible deviations of all outputs of a component, and so provides a description of how that component fails and reacts to failures elsewhere. At the same time, numerical data can be entered for the component, detailing the probability of internal failures occurring and the severity of output deviations. This data will then be used during the analysis phase to arrive at a figure for the unavailability of each top event. Once
done, the component can then be stored together with the failure data in a library, so that other components of the same type can use the same failure data or this type of component can be re-used in other models with the same failure data. This avoids the designer having to enter the same information many times.

For the specification of the components' failure modes (which are the effects by which the component failures are observed), a generic and abstract language was developed. There are different ways of classifying failure modes, e.g. by relating them to the function of the component, or by classifying according to the degree of failure – complete, partial, intermittent etc. In general, however, the failure of a component will have adverse local effects on the outputs of the component which, in turn, may cause further effects travelling though the system on material, energy or data exchanged with other components. Therefore in HiP-HOPS, effects are generally classified into one of three main classes of failure, all equally applicable to material, energy or data outputs. These are, namely, the omission of an output, i.e. the failure to provide the output, a commission of an output, i.e. a condition in which the output is provided inadvertently and in the wrong context of operation, and an output malfunction, a general condition in which the output is provided but at a level which deviates from the design intention. Since this classification adopts a functional viewpoint which is independent of technology, it could provide a common basis for describing component failures and their local effects. However, HiP-HOPS can work with any classification of failure modes as long as it is consistent from one component to the next.

Components do not only cause failures; they also detect and respond to failures caused by other components or transfer failures of other components. In addition to generating, mitigating, or propagating failures, they may also transform input failures to different types of output failure. For instance, a controller may be designed to respond to detected sensor failures by omitting any further output to ensure that hazardous control action is avoided. In this case, malfunctions at the input are intentionally transformed into omission failures. To capture those general patterns of behaviour of a component in the failure domain, manual analysis is performed at component level and focuses on the output ports through which a component provides services to other components in the system. In the course of the analysis, each output port is systematically examined for potential deviations of parameters of the port from the intended normal behaviour, which generally fall into the following three classes of failure:

(a) Omission: failure to provide the intended output at the given port
(b) Commission: unintended provision of output
(c) Malfunction: output provided, but not according to design intention.

Within the general class of malfunctions, analysts may decide to examine more specific deviations of the given output which, in most applications, will include conditions such as the output being delivered at a higher or lower level, or earlier or later than expected. As an example, Figure 6 shows the analysis of a two-way computer controlled valve. The figure shows the valve as it would typically be illustrated in a plant diagram and records the results of the local safety analysis of the component in two tables that define valve malfunctions and output deviations respectively.
## Valve Malfunctions

<table>
<thead>
<tr>
<th>Failure mode</th>
<th>Description</th>
<th>Failure rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocked</td>
<td>e.g. by debris</td>
<td>1e-6</td>
</tr>
<tr>
<td>partiallyBlocked</td>
<td>e.g. by debris</td>
<td>5e-5</td>
</tr>
<tr>
<td>stuckClosed</td>
<td>Mechanically stuck</td>
<td>1.5e-6</td>
</tr>
<tr>
<td>stuckOpen</td>
<td>Mechanically stuck</td>
<td>1.5e-6</td>
</tr>
</tbody>
</table>

## Deviations of Flow at Valve Output

<table>
<thead>
<tr>
<th>Output Deviation</th>
<th>Description</th>
<th>Causes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Omission-b</td>
<td>Omission of flow</td>
<td>blocked OR stuckClosed OR Omission-a OR Low-control</td>
</tr>
<tr>
<td>Commission-b</td>
<td>Commission of flow</td>
<td>stuckOpen OR Commission-a OR Hi-control</td>
</tr>
<tr>
<td>Low-b</td>
<td>Low flow</td>
<td>partiallyBlocked OR Low-a</td>
</tr>
</tbody>
</table>

**Figure 6 - Failure annotations of a computer-operated two-way valve**

In normal operation, the valve is normally closed and opens only when the computer control signal has a continuously maintained value of a logical one. Valve malfunctions include mechanical failures such as the valve being stuckOpen or stuckClosed, and blockages caused by debris such as blocked and partiallyBlocked. For each malfunction, the analysis records an estimated failure rate while the effects of those malfunctions on the output of the valve can be seen in a second table that lists output deviations.

This specification of failure modes is generic in the sense that it does not contain references to the context within which the valve operates. Failure expressions make references only to component malfunctions and input/output ports of the component. The failure behaviour described in these expressions has been derived assuming a simple operation that the component is expected to perform in every application (valve is normally closed unless the value of control signal is 1). For these reasons, the specification of Figure 6 provides a template that could be re-used in different models and contexts of operation, perhaps with some modifications, e.g. on failure rates, to reflect a different environment.

### 2.1.5.3 Synthesis Phase

As seen in Figure 6, component failure data relate output deviations to logical expressions that describe the causes of those deviations as component malfunctions and deviations of the component inputs. Each such expression is effectively a mini fault tree which links a top event (the output deviation) to leaf nodes, some of which may represent input deviations. When a component is examined out of system context, input and output deviations represent only potential conditions of failure. However, when the component is placed in a model of a system, the input deviations specified in the analysis can actually be triggered by other components further upstream in the model and the specified output deviations can similarly cause more failures further downstream.

This mechanism by which output failures of a particular class at one end of a connection trigger input failures of the same class at the other end results in a global propagation of failures through the system which may ultimately cause significant hazardous failures at the outputs of the system. Given a model of the system and the local safety analyses of its components, it is possible to capture this global propagation of failure in a set of fault trees. These fault trees are mechanically constructed by traversing the model and by following the propagation of failure backwards from the final elements of the design (e.g. electromechanical actuators) towards the system inputs (e.g.
material/energy resources, operators and data sensors). The fault tree is generated incrementally, as the local safety analyses of the components are encountered during the traversal, by progressively substituting the input deviations for each component with the corresponding output failures propagated by other components. Figure 7 illustrates the principle that underpins this process of fault tree synthesis. The figure shows a hypothetical motor and its starter circuit as a unit (M) that transforms electrical power provided by a power supply (PS) to mechanical power on a rotating axis.

![Figure 7 - Example system and fragments of local safety analyses](image)

The motor starter receives normal start and stop commands from a control computer (Controller). As a safety feature the controller is also connected to a sensor (LS) that monitors the load connected to the axis of the motor and when the measurement exceeds a specified load it issues a stop command to protect the motor. To illustrate the fault tree synthesis, the figure also provides a table that contains, for simplicity, only fragments from the local analyses of the Motor, the Power Supply, the Controller and the Load Sensor. For simplicity, deviations in this table refer to names of connections in the model rather than local names of component ports. Collectively, these deviations and their causes define the propagation of failures that result in an omission of mechanical power at the output of the motor.

The local analysis of the motor, for example, defines that this event can be caused by a failure of the motor, an omission of electrical power at the input, an omission of the start signal or, interestingly, a commission of the stop signal. The causes of some of those events can in turn be explored further in the local analyses of the components connected to the motor. For example, the analysis of the power supply defines that a failure of this component will cause an omission of electrical power. In turn, the analysis of the controller defines that an omission of the start signal will be caused by a controller failure while a commission of the stop signal can be caused by electromagnetic interference or in response to an abnormally high measurement of motor load. The analysis of the load sensor defines that the latter is indeed a plausible failure mode that can be caused in normal conditions of loading if the sensor is positively biased.

An overall view of the global propagation of failure in the system can be automatically captured by traversing the model and by following the causal links specified in the local safety analyses of the components that are progressively encountered during this traversal. The result of this process for the above example is the fault tree that is illustrated in Figure 8. Note that this mechanically synthesised fault tree records the propagation of failure in a very strict and methodical way. It starts from an output failure, the omission of mechanical power, and following dependencies between components in the model it systematically records other component failures that progressively contribute to this event. The logical structure of the tree is determined only by interconnections between the components and the local analyses of those components. This logical structure is straightforward and can be easily understood, unlike the structure of many
manually constructed fault trees which is often defined by implicit assumptions made by analysts. Note that although in this example the tree only incorporates OR gates, other logical symbols such as AND and priority AND gates would appear in the tree structure if such relationships were originally specified in some of the local analyses.

Figure 8 - Mechanically constructed fault tree for the example system

2.1.5.4 Analysis Phase

In the final phase, the synthesised system fault trees are analysed, both qualitatively and quantitatively, and from these results the FMEA is created. Firstly, the fault trees undergo qualitative analysis to obtain their minimal cut sets, which reduces them in size and complexity. This is done using a mixture of classical logical reduction techniques, which usually means applying logical rules to reduce complex expressions, and more modern techniques, such as the use of Binary Decision Diagrams (BDDs), to break down the tree into a simpler form. BDDs are graphs that represent the paths of failures through the fault tree, and are much faster and more efficient than classical methods, but unfortunately they cannot be used in all situations. Once the minimal cut sets have been obtained, they are analysed quantitatively, which produces unavailability values for the top events of each fault tree.

The last step is to combine all of the data produced into an FMEA, which is a table that concisely illustrates the results. The FMEA shows the direct relationships between component failures and system failures, and so it is possible to see both how a failure for a given component affects everything else in the system and also how likely that failure is. However, a classical FMEA only shows the direct effects of single failure modes on the system, but because of the way this FMEA is generated from a series of fault trees, the FTS tool is not restricted in the same way, and the FMEAs produced also show what the further effects of a failure mode are; these are the effects that the failure has on the system when it occurs in conjunction with other failure modes. Fig. 5.3.5 shows this concept.
In Figure 9, F1 and F2 are system failures, and C1 – C9 are component failures. For C3, C4, C6 and C7, there are no direct effects on the system – that is, if only one of these components fail, nothing happens. However, they do have further effects; for example, C3 and C4 both occurring in conjunction will cause F1 to occur. The FMEAs produced, then, show all of the effects on the system, either singly or in combination, of a particular component failure mode. This is especially useful because it allows the designer to identify failure modes that contribute to multiple system failures (e.g. C5 in the example). These common cause failures represent especially vulnerable points in the system, and are prime candidates for redundancy or extra reliable components.

2.1.5.5 Optimisation Phase

HiP-HOPS analysis may show that safety, reliability and cost requirements have been met in which case the proposed system design can be realised. In practice, though, this analysis will often indicate that certain requirements cannot be met in which case the design will need to be revisited. This indeed is a problem commonly encountered in the design of reliable or critical systems. Designers of such systems usually have to achieve certain levels of safety and reliability while working within certain cost constraints. Design of course is a creative exercise that relies on the technical skills of the design team but also on experience and successful earlier projects. So the bulk of design work is creative. However, some further automation could assist the decision on the selection among alternative components or sub-system designs as well as on the level of replication of components in the model that is required to ensure that the system ultimately meets its set safety and reliability requirements with minimal cost.

Figure 9 - The conversion of fault trees to FMEA

The FMEAs produced, then, show all of the effects on the system, either singly or in combination, of a particular component failure mode. This is especially useful because it allows the designer to identify failure modes that contribute to multiple system failures (e.g. C5 in the example). These common cause failures represent especially vulnerable points in the system, and are prime candidates for redundancy or extra reliable components.
A high degree of reliability and safety can often be achieved by using a more reliable and expensive component, an alternative sub-system design or by using replicated components or subsystems to ensure that functions are still provided when components or subsystems fail. In a typical design though, there are many options for substitution and replication at different places in the system and different levels of the design (component/subsystem). It may be possible for example to achieve the same reliability by substituting two sensors here and three actuators there, or by replicating a single controller, a control subsystem etc. Different solutions however will lead to different additional costs, and the question here is which is the optimal solution, one for example that achieves a certain degree of reliability & safety with the minimum additional cost. Because the design options for replication in a non-trivial design are typically too many to consider it is virtually impossible for designers to address the above questions systematically, so people rely on intuition, or on evaluation of a few different design options. Some automation in that area would therefore be useful to designers in area.

This has been achieved by combining work on automated safety analysis with recent advances in design optimisation. More specifically, genetic algorithms are employed in order to progressively “evolve” an initial design model which does not meet requirements to a design where components and sub-system architectures have been selected and where replicas have been allocated in a way that minimises cost while achieving given safety and reliability requirements. In the course of the evolutionary process, the genetic algorithm typically generates populations of candidate designs which employ user-defined alternative implementations for components and subsystems as well as standard replication strategies. These strategies are based on widely used fault tolerant schemes such as hot or cold standbys and n-modular redundancy with majority voting. For the algorithm to progress towards an optimal solution, a selection process is applied in which the fittest designs survive and their genetic make up is passed to the next generation of candidate designs. The fitness of each design relies on cost and reliability. To calculate fitness, there must be ways in which to automatically calculate those two elements. An indication of the cost of a system can be calculated as the sum of the costs of its components (although for more accurate calculations life-cycle costs should also be taken into account such as production, assembly and maintenance costs). However, while calculation of cost is relatively easy to automate, the automation of the evaluation of safety or reliability is more difficult as conventional methods rely on manual construction of the reliability model (e.g. the fault tree, reliability block diagram or the FMEA). HiP-HOPS, though, automates the development and calculation of the reliability model, and therefore facilitates the evaluation of fitness as a function of reliability (or safety). This in turn enables a selection process through which the genetic algorithm can progress towards an optimal solution which can guarantee the required safety and reliability at minimal cost.

HiP-HOPS is supported by a mature tool which can perform system safety analysis and architectural optimisation. The tool has been developed as an independent plug-in and has so far been interfaced to Matlab Simulink and Simulation X.

2.1.5.6 Temporal Analysis in HiP-HOPS using Pandora

HiP-HOPS relies on fault trees as the basis for its analysis of systems: components are annotated with expressions representing small fault trees, the system fault trees are constructed by joining these together, and the FMEA is created from the cut sets of these fault trees. FTA is not without its own faults, however. In particular, FTA struggles with the representation and analysis of time, or at least system behaviour that depends on time in some way. Fault trees model the static causal relationships between individual components in a system and the effect their failure has on the system as a whole, either singly or in combination with other failures. For many systems this is sufficient, but for some this model is too simplistic to capture the full failure behaviour. For example, some systems are capable of reacting to and recovering from certain failures, perhaps by activating a standby component or some other corrective measure; in other systems, the potential failure behaviour varies according to what state the system is in. These are dynamic systems, and this dynamic behaviour is difficult to model accurately using FTA. As systems
become increasingly dynamic and correspondingly more complex, it becomes ever more important to find ways of overcoming this deficiency in FTA.

As a simple example of a dynamic system, consider the generic standby redundant system in Figure 10 [18].

![Figure 10 - Simple triple redundant system](image)

This system is generic in the sense that components A, B and C could be any input, control or actuating device. A is the primary component that takes input from outside the system (labelled "I"), performs some function on it, and feeds the result to the system output (D). S1 is a monitoring sensor that detects any omission of output from A and activates the first standby component, B. Similarly, S2 is another sensor that detects a failure of B and activates the second backup, C. In theory, all three components A, B, and C must fail for the whole system to fail – the failure of one or two of the three can be tolerated. Performing a classical fault tree analysis on this system seems to confirm this, showing that the system only fails if:

1. There is no input at I.
2. All of A, B, and C fail.
3. Both A and S1 fail.
4. All of A, B, and S2 fail.

Each of these four events or combinations of events is sufficient to cause the whole system to fail. However, this is a rather naïve view of things; in reality the situation is more complex than this. Consider results 3 and 4: what if one of the sensors fails after the other components? For example, in case 3, a failure of A and S1 will cause system failure. Yet if S1 fails after A, then it does not lead to a system failure because S1 has already performed its function (to detect failure of A) and activated standby component B. Once B has been activated, failure of S1 is irrelevant - it can no longer have any effect on the functioning of the system. Similarly, in case 4, if S2 fails after both A and B have failed, then the second backup component C will have been activated, and there is no system failure. Again, S2 will have served its purpose, and any subsequent failure has no effect on the rest of the system, which is now operating on component C alone. Both cases 3 and 4 are therefore unnecessarily pessimistic: they suggest that a failure of the sensor will always cause the system to fail, when in fact this is only true in certain situations and is dependent upon the order in which the components fail.

Conversely, since the sensors work by detecting an omission of output, if the first standby component B fails before A, then S2 would never detect a cessation in output from B, because it would never be activated; C therefore remains unused. This means that the second case, failure of all of A, B, and C, is then a dangerously optimistic result, since B failing before A is sufficient to cause the system to fail, regardless of the status of component C. Classical fault tree analysis is unable to model this type of dynamic behaviour and would therefore fail to accurately capture the full failure behaviour of this system. FTA models only the effects of the occurrence of faults without
also taking into account the effect that the sequence of those faults can have. Instead, the results obtained are either unnecessarily pessimistic or excessively optimistic.

This raises an important question: if a system with just five components can have such pronounced inaccuracies, how far can we trust an analysis of a system with hundreds of thousands of components, such as a nuclear power plant?

Pandora was created as a way of solving this problem [19]. It is an extension to traditional fault tree analysis that introduces new gates to represent dynamic failure situations and a new temporal logic with which to analyse these new gates. Pandora allows for the general representation and analysis of event sequences without introducing many new gates or overcomplicating the process of fault tree analysis with complex new semantics. To that end, it is based around a redefinition of the long-established Priority-AND gate; the aim is to remain as close as possible to the original philosophy of simplicity and flexibility that makes FTA popular, while also solving the ambiguities that plagued the original PAND gate.

Pandora relies on the definition of an event as given in the Fault Tree Handbook [20], which states that for the purposes of FTA, it is the occurrence of faults that matters, i.e. the transition of a component to a faulty state at a certain point in time, rather than the existence of the faulty state. Pandora thus allows the temporal ordering of events to be represented as part of the fault tree structure itself. If the occurrence of an event is instantaneous and can occur at most once, and this means there are only three possible temporal relations between any two events X and Y:

- **before** - X occurs first, Y occurs second
- **after** - Y occurs first, X occurs second
- **simultaneous** – X and Y occur at the same time

These three temporal relations can be represented abstractly by three temporal logic gates. The first gate is the PAND gate, used to represent "before", e.g. X PAND Y means X occurs before Y. The Priority-AND gate is defined informally as follows:

**Name:** Priority-AND gate

**Abbreviation:** PAND

**Symbol:** <

**Meaning:** The PAND gate specifies a sequence. It is true if all input events occur and they occur in order from left to right. It is false if any of the events occur out of sequence or if any occur simultaneously.

The PAND gate can also be reversed to represent "after", i.e. X PAND Y means both "X before Y" as well as "Y after X". In this way, the PAND represents both of the first two temporal relations in Pandora.

The third is represented by the Simultaneous-AND or SAND gate, which means that two or more events occur at the same time. It is defined informally as follows:

**Name:** Simultaneous-AND gate

**Abbreviation:** SAND

**Symbol:** &

**Meaning:** The SAND gate specifies that all of its inputs must occur and that furthermore, they all occur simultaneously. It is false if any do not occur or if any occur at a different time.

The third gate also represents the "before" relation, but in this case it is different. Whereas the SAND and PAND gates are both types of AND gate and thus require all input events to occur (and thus they are known as 'temporal conjunction gates'), the third gate – the Priority-OR, or POR gate
is based on the OR gate (and is thus 'temporal disjunction gate'). It specifies that one event must occur before any others, but it does not specify that those other events must occur. Thus:

Name: Priority-OR gate
Abbreviation: POR
Symbol: |
Meaning: The POR gate specifies priority, and is true if its left-most input occurs before any of its other inputs or if none of the subsequent inputs occur. It is false if any of the other input events occur before or at the same time as the left-most input, or if the left-most event does not occur at all.

All three gates are required for Pandora's temporal logic to integrate properly with the traditional Boolean logic that underpins fault trees. Because there are only three possible temporal relations between any two events, if two events occur, then one (and only one) of these relations must be true. This can be represented by the following expression:

\[ X \cdot Y \iff X < Y + X \& Y + Y < X \]

Namely, that if X and Y both occur, then either X occurred before Y, they occurred simultaneously, or Y occurred before X. This is known as the **Conjunctive Completion Law** and is one of the foundations of Pandora. There are three Completion Laws, the other two being:

\[ X + Y \iff X \mid Y + X \& Y + Y \mid X \]
\[ X \iff Y < X + X \& Y + X \mid Y \]

The first is the Disjunctive Completion Law and specifies that if at least one of two events occurs, then either both occurred simultaneously, or only one occurred, either before the other or on its own. The second is the Reductive Completion Law and shows how an event (in this case Y) can be totally redundant if it does not matter *when* it occurs in relation to another event (in this case X).

The first two Completion Laws can be better seen as a diagram:

![Disjunctive Completion Law Diagram](image)

**Figure 11 – Disjunctive Completion Law**

The shaded part represents the conjunction while the white part represents the extra parts (i.e. X alone and Y alone) required by a disjunction. The five subsections and combinations thereof can all be described by the temporal gates according to the Completion Laws.

There are many other temporal laws in Pandora, ranging from temporal versions of the Distributive, Associative, and Absorption Laws to other fully temporal laws. Amongst the most important are the Laws of Mutual Exclusion, which introduce the concept of *contradiction*:

\[ X < Y \cdot Y < X \iff 0 \]

This is the first Mutual Exclusion law and specifies that X<Y and Y<X cannot *both* be true. The Mutual Exclusion laws embody the principle that only one of the three basic temporal relations can be true at once, thus any combination of two or more is inherently contradictory. Because a contradiction is always false, it can be removed from the fault tree (e.g. X + 0 \iff X) or even render large parts of the tree false as well (e.g. X . 0 \iff 0). Some contradictions are harder to spot than others, however, and amongst the more subtle are the circular contradictions. For example:

\[ X < Y \cdot Y < Z \cdot Z < X \iff 0 \]

This expression does not immediately seem to violate Mutual Exclusion. To detect the contradiction, we first need to apply the Law of Extension, which states:
The Law of Extension reveals omitted temporal relations between events. In this case, if X is before Y and Y is before Z, then by extension, X must also be before Z. When applied to the circular contradiction, it shows that:

\[ X < Y \land Y < Z \iff X < Y \land Y < Z \land X < Z \]

Now there are plenty of violations of Mutual Exclusion and the expression can be seen to be a contradiction.

Laws like these form the basis of logical reduction in Pandora, making it possible to reduce a temporal fault tree (i.e. one containing temporal gates as well as normal gates) into minimal cut sequences, which are the smallest possible sequences of events capable of causing the top event of the fault tree. They can also be proved by means of temporal truth tables. Temporal truth tables are extensions of normal Boolean truth tables in the same way that Pandora is a temporal extension to normal Boolean fault trees. Just as Pandora makes it possible to specify the sequence of events as well as the occurrence of events, temporal truth tables (or TTTs) make it possible to specify the order in which events occur as well as whether or not they occur.

This is done using sequence values. In Pandora, the exact time at which an event occurs is not important – the only thing that matters is when it occurs relative to the other events, i.e. which comes first, which comes second, which comes last etc. Nor is it necessary to say exactly how long the interval between these events lasts. Sequence values are used instead as a way of representing this ordering information: they are abstractions of the time at which an event occurs. Furthermore, because Pandora models only instantaneous occurrences, sequence values can also be used for gates too.

To indicate a sequence value, the notation \( S(X) \) is used, where \( X \) is an event (or a gate). The sequence value is not the same as the truth value of an event, although they are closely linked. If an event or gate is false, it means it has not yet occurred. Because it has not yet occurred, it is given a sequence value of 0 (i.e. no sequence). If an event is true, it has occurred, and therefore has a sequence value greater than 0 to indicate when it occurred relative to the other events under consideration, e.g. 1 means it occurred first, 2 means second and so on. Two events can have the same sequence value, in which case it means they both occurred simultaneously. In this, sequence values can be likened to a race: the first person to cross the finishing line gets 1st place, the second gets 2nd place, and so on; if two or more people cross the finishing line, they get a joint position, e.g. joint 3rd place. Someone who did not finish the race would not get a place, i.e. 0.

This notation also makes it possible to distinguish easily between events that have occurred and are true, which will have non-zero sequence values, and events that have not occurred and are false, which have sequence values of zero. This is very similar to the customary notation for Boolean logic of using 0 for false and 1 for true, except now any non-zero positive integer represents true. Because it is possible in this way to represent both the temporal and logical values of events with just the sequence values, the \( S(\cdot) \) notation is often omitted in temporal truth tables (although it is possible to create truth tables in which the temporal and logical values are separate, in which case \( L(\cdot) \) gives the logical value while \( S(\cdot) \) gives the temporal value).

Below is a TTT showing the behaviour of the first Completion Law; note that the two bolded columns are the same, indicating (as in a normal truth table) that the expressions are equivalent.
To see how Pandora works in practice, it can be applied to the simple example system in Figure 10. To describe the failure behaviour of the system, we need to be able to explain that B is only activated after a failure of A – assuming S1 has not yet failed – and that C is only activated after a failure of B – again, assuming that the sensor (S2) has not yet failed. To do this properly we need to represent the sequence of events. Each component can be annotated with Pandora logic to describe how it fails and how it reacts to failure at its inputs. A can fail either as a result of internal failure (failureA) or due to a lack of input from I (Omission of input, or O-I). Either failure will result in an omission of output. Thus the failure behaviour of A can be described as:

\[ O-A = \text{failureA} + O-I \]

The failure behaviour of B is somewhat more complicated because it is dependent on A. We can use a special event, startB, to represent the trigger for B to be activated. startB occurs when O-A occurs, i.e. when A fails. The startB event is generated by sensor S1, which detects O-A. S1 can itself fail due to an internal failure, failureS1. Thus the behaviour of S1 would seem to be:

\[ O-S1 = \text{failureS1} \]
\[ \text{startB-S1} = O-A \]

However, this is a simplistic view, because as already explained, if the sensor fails after an omission of output from A has been detected, then it has no effect. Thus:

\[ O-S1 = \text{failureS1} < O-A + \text{failureS1} \land O-A \]
\[ \text{startB-S1} = O-A \]

Here the temporal gates have been used for the first time. An omission of S1’s signal is caused by its failure before or at the same time as an omission of output from A. In these cases, no start signal will be sent (and B will not be activated). Now we can also represent the failure behaviour of B. B will omit its output if it does not receive the start signal when A fails (e.g. due to a sensor failure) or if loses input or it fails internally once it has been activated. Thus:

\[ O-B = O-S1 + \text{startB-S1} < (\text{failureB} + O-I) \]

However, again the situation is not so simple. This is because if B does not receive its start signal, or fails dormant before the signal is received, then it will never activate. This actually causes an undetectable omission, whereas the failure of B once it has started causes a detectable omission, i.e. because the output of B starts and then stops, its cessation can be detected. A revised version would be:

\[ Odet-B = \text{startB-S1} < (\text{failureB} + O-I) \]
\[ Ound-B = O-S1 \]
This is now a more accurate reflection of the failure behaviour. A detectable omission happens once the component has been activated whereas an undetectable omission arises if the component does not activate before failure or never receives its activation signal.

The expressions representing the failure behaviour of the system can be combined into a fault tree by means of substitution, e.g. substituting \((O-I + \text{failureA})\) for all instances of \(O-A\). The resultant expression is:

\[
O-D = \text{failureS1} < \text{failureA} + O-I \\
+ \text{failureS1} \& \text{failureA} + O-I \\
+ \text{failureB} + O-I < \text{failureA} + O-I \\
+ \text{failureB} + O-I \& \text{failureA} + O-I \\
+ \text{failureS2} < ((\text{failureA} + O-I) < \text{failureB} + O-I)) \\
+ \text{failureS2} \& ((\text{failureA} + O-I) < \text{failureB} + O-I)) \\
+ \text{failureC} + O-I).(\text{failureA} + O-I) < (\text{failureB} + O-I))
\]

This expression can then be reduced by applying temporal laws. For example:

\[
(failureB + O-I) \& (failureA + O-I)
\]

is expanded using a SAND distributive law, \((Y+Z) \& X \Leftrightarrow (X\&Y).Y+Z + (X\&Y).Y|Z + (X\&Z).Z|Y\), to become:

\[
\begin{align*}
(failureB + O-I) \& (failureA + O-I) \\
\Leftrightarrow (failureA + O-I) \& O-I \& failureB + \\
(failureA + O-I) \& O-I \& failureB \|
\end{align*}
\]

These can then be minimised, e.g. \(O-I \& O-I\) reduces to just \(O-I\), while

\[
O-I \& failureA \| O-I
\]
is a violation of Mutual Exclusion and becomes 0. Once fully reduced, the expression becomes just:

\[ O-I \land \text{failureA} \land \text{failureB} \]

\[ + O-I \lor \text{failureA} \land O-I \lor \text{failureB} \]

\[ + O-I \land \text{failureA} \lor \text{failureB} \]

\[ + O-I \land \text{failureB} \lor \text{failureA} \]

\[ + \text{failureA} \land \text{failureB} \lor O-I \]

Once fully reduced, there are eight minimal cut sequences for the fault tree, describing the possible ways that the top event (omission of output) can occur:

1. 0-I
2. failureA & failureB
3. failureB < failureA
4. failureS2 < failureA < failureB
5. failureA < failureS2 < failureB
6. failureS2 & failureA < failureB
7. failureA < failureS2 & failureB
8. failureC . failureA < failureB

In other words:

- an omission of input, which is a common cause failure as all of A, B, and C will fail;
- a failure of A and B at the same time, meaning B will never activate and will cause an undetectable omission of output;
- a dormant failure of B before A, meaning B will never activate and will cause an undetectable omission of output;
- a failure of sensor 2 before component A before a failure of B, meaning that C will never be activated;
- a failure of A before sensor 2 before a failure of B, meaning that C will never be activated;
- a simultaneous failure of sensor 2 and component A before a failure of B, meaning that C will never be activated;
- a failure of A before a simultaneous failure of sensor 2 and B, meaning that C will never be activated;
- a failure of all three components, assuming A fails before B (and thus B gets activated).

Thus Pandora can reveal much more information about the precise failure behaviour of dynamic systems than ordinary fault tree analysis. This is particularly important for the analysis of safety-critical systems, which frequently display dynamic behaviour as part of fault tolerance strategies. Because Pandora is designed to integrate with existing Boolean logic, it can be more readily used in existing tools such as HiP-HOPS, adding additional dynamic FTA capabilities. However, at present quantitative analysis with Pandora is not possible, though other techniques exist which are
capable of this, such as the Dynamic Fault Tree (DFT) approach [21]. DFTs counterbalance Pandora well: they are capable of quantitative but not qualitative analysis while Pandora is capable of qualitative but not quantitative analysis.

2.1.6 AADL

The Architecture Analysis and Design Language (AADL) is an Architecture Description Language (ADL) that has been developed under the auspices of the International Society of Automotive Engineers (SAE), Avionics Systems Division (ASD), and committee of Embedded Computing Systems. AADL is therefore an SAE standard particularly targeted to the design and analysis of hardware and software architectures of performance-critical real-time systems.

The AADL language has several capabilities, including the structural description of the system as an assembly of software components mapped onto an execution platform, functional description of interfaces to components e.g. for data inputs and outputs, and performance description of critical aspects of components for e.g. timing. Recently, the language has been extended with an error model annex, which complements the SAE standard with capabilities for dependability modelling.

This section of the document provides an overview of AADL: its components, connectors, system and properties, as well as the capabilities for dependability modelling in AADL, discussed in association with some examples. The emphasis is therefore on error modelling of components, reuse of error models, modelling and handling of error propagations, interactions between error models, and abstraction of error models.

2.1.6.1 The elements of AADL

This section presents the key specification elements of AADL. For instance, a component type declaration defines features (the component’s interface elements) through which interactions with other components may occur. A component implementation declaration defines the internal structure of the component under study. The latter may consist of subcomponents, connections between subcomponents, properties such as occurrence of an event, or properties for handling error propagations amongst other subjects.

2.1.6.1.1 Components

The AADL component abstractions are distributed amongst three groups which are application software, execution platform, and aggregates [22]. The first abstraction, and as shown in the top left part of Figure 12 below, consists of threads (active components), thread groups (logical organisations within a process), processes (protected address spaces), data (static) and subprograms (callable pieces of code). The second abstraction, and as shown in the right part of the figure, insulates details about the processor (scheduler & executor), memory (storage for code and data), devices (sensors, actuators and so forth) and buses (interconnections). The third abstraction, and as shown in the bottom left part of the figure, is the overall system, i.e. integration of a combination of software and/or hardware components.
Moreover, the language has the capability to describe connections, i.e. components’ interactions, and also shows how software components are allocated to execution platform components.

2.1.6.1.2 Connectors

Connectors describe communication channels between components and coordinate their interactions. Connections are represented as the actual linkage between components. Figure 13 describes, amongst other elements, different types of communication interfaces between components often called ports. Ports are for communicating data, events, or data events. An important purpose of this categorisation of ports is to address interaction consistency, i.e. to match port types with data communicated through the ports. Therefore, it stimulates modelling of systems as interacting components with specific semantics and bindings.

Communication can either be with queues (event or event-data) or without queues (data). Thus data ports are used for synchronous communications of state data, either immediate or delayed; both are represented on the right side of the figure. Event ports, however, are used for events (queues) and event data ports are used for queued message data (message passing).

Some ports may be input ports, output ports or even both. For instance, assuming a thread which stores data and an event called flush through an input event port in event port, to flush some raw data through an in event data port. Similarly, we may assume another thread to acquire data with its corresponding acquire event through the out event port, and with an output event data port out event data port for some raw data to be acquired. Moreover, another example that consists of a thread called Data Manager which runs by an event called start through its in event port.

In addition to that, Figure 13 represents a port group as the grouping of ports which is treated as a single unit outside a component [114]. This allows collections of ports to be connected with a single connection. However, the ports of a port group can be accessed individually inside a component. Interactions between components may also be represented by subprogram call sequences.
Figure 13 - Connections and Port Graphical Representations in AADL

Flows represent the logical information that passes through components and connections in AADL may be used to specify a mode change.

2.1.6.1.3 System

A system is made up of components and connectors and may also be hierarchical. In that sense, components and connectors may represent subsystems with their internal architectures. Moreover, AADL has a package configuration element, as represented in Figure 12, which groups components with their connections into modules. This is to facilitate reuse of elements, components and connectors for instance, in the definition of a system.

2.1.6.1.4 Properties

In general, the properties of ADLs represent semantic information about a system, its subsystems, and their components. These properties are the basis of information provided for analysis tools to determine the behaviour and performance of the system being modelled. As an example, the calculation of overall system throughput and latency which is based on performance estimates of each component and connector [114].

AADL provides a predefined set of properties that apply to components and connectors. For example, the execution time of a thread for each component category. Moreover, AADL includes support for user defined property sets to provide additional information about a system or a component. For instance, properties to specify changes of the error state of a component due to error events or error propagations.

The next figure summarises the elements of AADL and from type and implementation perspectives.
A component type defines the interface elements (features shown in the figure above), a component implementation defines the internal structure (subcomponents, connections etc), the packages serve to group AADL elements, and property sets with annex libraries enable the customisation of the specification for domain specific requirements and for error modelling. Further details may be found in [22].

2.1.6.2 Overview of the dependability modelling approach

This section describes modelling in AADL with an emphasis on modelling dependability. Dependability is defined for this purpose as the ability of a system to continue to produce the desired service to the user when the system is exposed to undesirable conditions [23].

The dependability model consists of both architecture modelling and error modelling. Architecture modelling is the architecture description of the components with their connections and interactions, also referred to as nominal architecture. It describes both the structural and some aspects of the behavioural designs of AADL elements but without faults or considerations of malfunctions of an element (or of a combination of elements). In contrast, error modelling addresses the behaviour of components in the presence of, on the one hand, internal faults and repair events and, on the other hand, external propagations from the component environment. Thus, only components which are associated with error models, including their connections, are part of the AADL dependability model.

2.1.6.2.1 The Error Model

Error modelling provides a means of capturing potential errors, failure behaviours, propagations, and hence provides necessary support for analysing system safety and reliability. Such models are normally derived by extending the nominal models with additional information characterising errors and failure behaviours.
AADL error models are described in the AADL error model annex, which was created by the AADL working group, and is intended to support qualitative and quantitative analysis of dependability attributes [24]. The AADL error model annex defines a sub-language that can be used to declare error models within an error annex library.

An AADL error model consists of a model type and, at least, one error model implementation. It is a state machine that can be associated with an AADL element, i.e. component or connection, in order to describe its behaviour in terms of logical error states in the presence of faults. Error models can be associated with hardware components (processor, memory, device, and bus), software components (process, subprogram, data, thread, and thread group), composite components (system) and connections.

Based on the common foundation of Architecture Description Languages (ADLs), AADL components represent the elements of a system. Hence, systems may be represented as collections of components, hierarchies of components, or systems of systems. With this in mind, the scope of an AADL error model extends from system to subsystem to component. In that sense, the system error model is a composition of the error models of its components or subsystems. It is meant to capture hazards at system level, risk mitigation architecture at subsystem level (i.e. intermediate level), and to capture failure modes and effects analysis model at component level. This expansion reflects propagations of errors between components which are based on the component dependencies.

Moreover, the error models may be reusable to automatically regenerate safety and reliability models, e.g. when making alterations of the architecture specifications. There are two kinds of reusable error models: the basic model and the derived model. The basic error model consists of a declaration of error states, for a component or a connection, and properties to specify how error state changes due to error events and propagations. However, the derived error model is the definition of the error state of a component in terms of the error states of its subcomponents.

The example below shows the definition of a basic error model which is divided into a type and an implementation. There may be more than one implementation.

![Error Model Definition for a Component without propagation](image-url)

Figure 15 – Error Model Definition for a Component without propagation
error model Example1
features
ErrorFree: initial error state;
Failed: error state;
Fail, Repair: error event;
CorruptedData: out error propagation
  {Occurrence => fixed 0.8};
end Example1;

error model implementation Example1.basic
transitions
ErrorFree- [Fail] ->Failed;
Failed- [out CorruptedData] ->Failed;
Failed- [Repair] ->ErrorFree;
Properties
Occurrence => poisson 1.0e-3 applies to Fail;
Occurrence => poisson 1.0e-4 applies to Repair;
end Example1.basic;

CorruptedData of the example in Figure 15 above is an output propagation. However, an alteration of the example to model an input output error propagation, for instance CorruptedData, would be as follows:

![Figure 16 - Error Model Definition for a Component with in out propagation](image)

error model Example2
features
ErrorFree: initial error state;
Failed: error state;
Fail, Repair: error event;
CorruptedData: in out error propagation
  {Occurrence => fixed 0.8};
end Example2;

error model implementation Example2.basic
transitions
ErrorFree- [Fail] ->Failed;
Failed- [out CorruptedData] ->Failed;
Failed- [Repair] ->ErrorFree;
ErrorFree- [in CorruptedData] ->Failed;
Properties
Occurrence => poisson 1.0e-3 applies to Fail;
Occurrence => poisson 1.0e-4 applies to Repair;
end Example2.basic;

The type of the error model consists of the declaration of the error states (i.e. ErrorFree and Failed) and events (i.e. Fail and Repair). An error state is simply a state that has meaning for error modeling purposes. It does not necessarily imply faulty behaviour, for instance, the initial error state (i.e. ErrorFree) is often assumed to be a fault-free state. However, an error event is any internal intrinsic event that changes the error state of a component. The implementation part of the
example above shows a transition from the ErrorFree error state to the Failed error state when the error event Fail occurs. The component becomes again ErrorFree if the transition is triggered by the Repair event. The component sends the CorruptedData error propagation.

The Occurrence properties for error events and error propagations specify the arrival rate (i.e. the poisson keyword) or the occurrence probability (i.e. the fixed keyword). Those properties can be declared in the error model type and in the error model implementation. The occurrence values in the error model type are considered as default values. In that sense, an occurrence value in the error model type can be replaced by a component specific value whose component is associated with the error model.

The below elucidates an example where a component-specific property value overrides a default value and, also explains how error models are meant to be reusable. The error model described earlier is part of an annex library even so, several error model definitions are declared between the constructs annex Error_Model {**} and **}; and as shown below:

```plaintext
package My_ErrorModels
  public
  annex Error_Model {**
    error model Example1
      ...
    end Example1;
  
  error model implementation Example1.basic
    ...
  end Example1.basic;

  error model Example2
    ...
  end Example2;

  error model implementation Example2.basic
    ...
  end Example2.basic;
  **};
End My_ErrorModels;
```

The part below shows an example where a previously defined error model is being reused and also explains how a component-specific property value overrides the default value declared in the error model type. This is done by defining a system called computer with an implementation called computer.personal, then associating an error model of the annex library (i.e. Example1) with its implementation (i.e. Example1.basic).

```plaintext
system computer
  end computer;

system implementation computer.personal
  annex Error_Model {**
    Model => My_ErrorModels::Example1.basic;
    Occurrence => fixed 0.9 applies to error CorruptedData;
  **};
End computer.personal;
```

The error model Example1 is associated with any component that is an instance of the component implementation containing the Model property. If an error model is not meant to be reusable then,
the *Model* property allows its definition without placing it into an error model annex library. In the last case, the error model cannot be instantiated somewhere else.

Furthermore, the example above expresses an outgoing propagation *CorruptedData* which occurs at higher probability (i.e. 0.9) than the default probability (i.e. 0.8) which was specified as part of the error model type definition declared in the library. The association of a property value with an error event or propagation – in this example, of a component–specific occurrence property value with the outgoing propagation *CorruptedData*, realised using the clause **applies to error**.

Similarly, error model property values can also be specified for subcomponent instances, and connections as well, and as described below.

```
system computer
end computer;

system implementation computer.personal
subcomponents
  CPU: processor Intel.DualCore;
  RAM: memory SDRAM;
  FSB: bus FrontSideBus;
annex Error_Model {**
  Model => My_ErrorModels::Example1.basic applies to CPU;
  Occurrence => fixed 0.9 applies to error CPU.CorruptedData;
**};
end computer.personal;
```

The Example1.basic implementation of the annex library error model Example1 is, in the above, associated to the CPU component by using the clause **applies**. However, the clause **applies to error** associates a higher probability of the outgoing error propagation *CorruptedData* to the CPU subcomponent of a computer system. This occurrence property value overrides the property value, of the same feature, declared either in the type or in the implementation of the error model defined in the library. In the case of deeper embedding of subcomponents, we use a dot-separated sequence of subcomponent names followed by the name of the feature.

An error state declared in an error model definition can represent for e.g., a failure mode identified in a failure mode and effects analysis i.e. FMEA, or a hazardous state identified in a hazard analysis; and depending on the dependability analysis to be performed. Some dependability analyses do not require probabilistic dependability measures. Therefore, they do not require the definition of occurrence properties, which justifies the optional character of the occurrence property declarations.

Although called error state, AADL error states can represent error-free states as well as error states. Similarly, AADL error events may represent repair events as well as fault events.

2.1.6.2.3 Derived error model

A derived error model means a component error model such that the component is in its error state under certain conditions. Those conditions are, on the one hand, described in terms of error states of subcomponents. On the other hand, they are described in terms of error states or error propagations of connections or connected components.

The process of determining an error state, of a derived error model associated with a component, is also called a derived error state mapping. In that sense, subcomponents’ error states, error
states of components which are connected to input ports and thus, incoming error propagations. All those map to a derived error state of the component under study. The following shows the derived state mapping to an error state of a component with two subcomponents: hardware (HW) and software (SW).

Derived State Mapping =>
   ErrorFree when HW[ErrorFree] and SW[ErrorFree],
   Failed when others;

The keyword others should only be used in the final state-mapping rule. It is the rule which will determine the current error state of the component and, if no preceding rule has been evaluated to true. This means evaluations of the state mapping rules are in the order of their declarations. Evaluations cease at the first when clause evaluated to true.

2.1.6.2.4 Error propagations

Interactions between components come at the expense of failing components compromising others. Error models, which are associated with interacting components, process possible error propagations between those components. Example1 and Example2, given in this document, show that error models are enhanced by the declaration of error propagations. The reference to error propagations consists of error state transitions.

A component which may propagate an error declares in its error model an out error propagation. The error propagation occurs when - it is named in an error state transition and, - the current error model state of the component is the origin of that transition. Moreover, the error propagation occurs according to the specified probability assigned to the propagation occurrence property [25]. The mapping of an out error propagation of one component to an in error propagation of an impacted component is determined by name matching or specified as propagation guards for some system components.

Examples 1 and 2, of Figure 15 and Figure 16 respectively, declare an error event fail and an error propagation CorruptedData with different names. This means that an error state of the component, which is associated with any of those error models, may not always be visible, i.e. being an incoming error, to other components, or it is but with a delay.

If we assume that the error state of that component is always visible to other components then, the occurrence probability of the error propagation should be 1. We can also, in that case, declare the fail error event as an error propagation but it wouldn’t be recommended for reuse purposes. In that sense, for that same case, it is preferable to have an error event with a different name and to assign a probability of 1 to the occurrence of the error propagation. Consequently, the error model still remains reusable for other cases by just overriding the value assigned to the probability of occurrence of the error propagation.

---

1 For instance, Figure 15 or Figure 16.
2 A component associated with an error model, which declares an in error propagation, knows its incoming error by its name.
3 They are propagation properties on how to handle incoming propagations.
4 Subsection 2.1.6.2.4 presents the guard properties.
5 The probability assigned to the occurrence property of the error propagation is, for e.g., \( p \), with \( p \neq 1 \).
According to [25], AADL error propagations are discerned by the definition of dependency rules. Those rules are determined by, on the one hand, the components’ interdependencies in the architectural model and, on the other hand, they are determined by inheritance and thus, known as inheritance rules. In that sense, when a component or a connection does not have an error model associated with it. Then, we investigate any existence of error models associated with subcomponents or associated with one of the parent components, and in order to process that propagation.

Dependency rules for error propagations:

The first category of rules for error propagations, i.e. dependency rules, extends across dependencies that are issued from sharing hardware, from application interactions, from hardware interactions, and also dependencies issued from special cases in system architecture. An example of a special case in system architecture is such that a subcomponent may propagate errors to every subcomponent of the same process. Also, for e.g., a process that, possibly, propagates errors to every other process that is bound to any common resource. In the latter examples, exceptions are made for the case of, for instance, processes that are partitioned from each other on all common resources. Sharing hardware dependencies apply to a case, for example, where a processor may propagate errors to every thread that is bound to that processor. Also, another example of a bus component that may propagate errors to every connection routed through that bus.

Furthermore, application interactions dependencies apply to a case, for e.g., where an application component may propagate errors to each of the data component it has access to, and through “provides” and “requires” data access declarations. Another rule of this kind, i.e. application interactions, is such that error propagations may occur from a component to every connection from any of its out ports. Or else, from a connection to every component having an in port to which it connects. In other words, an application component can affect any connected component through its outgoing connections.

Concerning hardware interactions rules, those are issued from interconnections between execution platform components, and through shared access to buses. In that sense, error propagations may occur from a component to each bus that is accessed through bus access declarations, i.e. provides and requires. Or else, from a bus to each accessing component, i.e. bidirectional flow.

Figure 17 shows that error propagation, i.e. red arrows, between error model instances, i.e. represented as state machines inside the components, may occur via: application components interactions, i.e. the upper part, through a port connection and from an out port to an in port, and in conformance with the dependency rules; hardware interactions, i.e. the lower part which shows an error propagation from a processor 1 to a processor 1 through a bus connection; and finally, apart from possible error propagations between interconnected hardware components,

---

6 Provides and requires are data access features, i.e. declarations to access shared data. Also, they may be used for bus access declarations and connections.

7 Application interactions dependency rules state possible propagations from a component to any component through its outgoing connections, i.e. unidirectional flow.

8 Hardware interactions dependency rules state possible propagations from a component to each accessed bus. Also, from a bus to each accessing component, i.e. bidirectional flow.
propagations may also occur from a hardware component to a software component which is binding to it\(^9\), i.e. upward pointing arrows.

![Diagram](image.png)

**Figure 17 - Error Propagations between Error Model Instances [25]**

Inheritance rules for error propagations:

Inheritance rules bridge the gap between dependency rules for error propagations and situations such that a component, hardware or software, does not have an error model associated with it. This might be the case of error propagation from a processor, with an associated error model, to a thread for e.g., and that is bound to that processor, but does not have an associated error model. This case is defined by dependency rules in the category called “sharing hardware”.

More generally, if dependency rules define error propagation *out of / in to* a component that does not have an error model associated with it but, does have subcomponents with error models. Then error propagation occurs *out of / in to* subcomponents error models, and follows dependency rules of the component. Similarly, if dependency rules define error propagation *out of / in to* a component that does not have an error model associated with it but does have a parent, i.e. a hierarchically contained component with an error model. Then error propagation occurs *out of / in to* the parent's error model, and follows dependency rules of the component.

Moreover, when a connection does not have an associated error model, then propagations are passed from any source of that connection to all ultimate destinations. Also, if a shared data component does not have an error model associated with it, then propagation occurs to all components that share access to that data component. Finally, errors never propagate from an error model instance to itself. For further details about error propagation inheritance rules, we refer to [25].

---

\(^9\) Shared hardware dependency rules state possible unidirectional propagations from hardware components to software components binding to those.
Managing incoming error propagations based on guard properties:

An error propagated out of a component may impact another component. The affected component is associated with an error model declaring an in error propagation with a matching name (i.e. same name as for the out error propagation). Alternatively, an error propagated out of a component may be observed by another component which will not transit into an error state. In that sense, the outgoing error propagation is named in the guard specifications declared in the error model instance of the observing component. There are two kinds of guard properties to handle error propagations: the Guard_In property, and whose role is to handle error propagations at the error receiver level, and the Guard_Out property, and whose role is to mask or to pass error propagations to other components.

Guard_In property

Guard_In property allows [25] conditional or unconditional mapping of the name of a propagation and an error state declared in the sender error model instance to a propagation name declared in the receiver error model. If it is conditional then, it might be the mapping of a set of incoming propagations from connected components to a single, or a set of, in error propagation(s). Moreover, Guard_In property allows the conditional masking of incoming propagations. The property evaluates when error propagations occur into a components through, for instance, in data port, in event port or in event data port. Furthermore, Guard_In property evaluates when propagations occur through other features such as for e.g. data access features i.e. requires and provides. Each component feature can only have one Guard_In property.

The following figure consists of a Guard_In property which allows the mapping of incoming propagation (Outprop in the figure), i.e. outgoing from the connected component, or error state of the connection or connected component, i.e. the encircled s in black colour in the figure, or even the error state of the component whose error model instance declares the Guard_In property, i.e. in red colour in the next figure, and every element which feeds the Guard_In is represented with a double-line arrow. The mapping will result into an in propagation (Inprop in the figure) or a masked propagation or error state.

![Guard_In Property for Rules that handle Incoming Propagations](image)

The example below shows an example of a use of a Guard_In property. in_propagation_1 and in_propagation_2 are incoming propagations declared in the error model instance of the receiving component (i.e. whose error model declares the Guard_In property). the when clause represents the guard expression for an evaluation. The result of an evaluation determines either a mapping to an incoming propagation name or simply masking the propagation.

The first rule following the Guard_In property states an unconditional mapping of the outgoing error propagation (syntactically braketed) from the port port_name_1 to an incoming error propagation named in_propagation_1.
The second rule states a conditional mapping to an incoming error propagation named \textit{in\_propagation\_2} when each of the next conditions evaluate to true. First condition, the error model instance associated with the feature \textit{port\_name\_2} is in its \textit{error\_state\_1} (also syntactically bracketed) and second, the error model instance associated with the feature \textit{port\_name\_3} is in its \textit{error\_state\_2}. This means, if both conditions evaluate to true together then an incoming error propagation is named.

The third rule expresses the masking of the incoming error propagation and when any of the following conditions evaluates to true. First condition, the error model instance associated with the feature \textit{port\_name\_2} is in its \textit{error\_state\_2} and, the second condition is such that the error model instance associated with the feature \textit{port\_name\_3} is in its \textit{error\_state\_1}.

The rules cited above are all related to managing incoming propagations through the feature \textit{port\_name\_1}.

\begin{verbatim}
Guard\_In =>
    in\_propagation\_1 when port\_name\_1[out\_propagation\_1],
    in\_propagation\_2 when port\_name\_2[error\_state\_1]
        and port\_name\_3[error\_state\_2],
    mask when port\_name\_2[error\_state\_2] or
        port\_name\_3[error\_state\_1]
    applies to port\_name\_1;
\end{verbatim}

Example 3

For further details about the Guard\_In specification, e.g. the logical operators such as 'and', 'or', 'ormore', 'orless' and 'not', we refer to [25].

\textbf{Guard\_Out property}

Similarly, a Guard\_Out property allows handling error propagations but, in terms of pass-through specifications of those propagations such as:

- unconditionally pass error propagations from different sources as outgoing propagations;
- similar to previous previous but under some conditions, and including the treatment of some error states as outgoing propagations;
- conditionally mask incoming propagations from different sources.

This property may apply, for instance, to outgoing ports for any of data, event or event data; also, it may apply e.g. for provides data access features. The property is evaluated when error propagations occur through some specified features such as an \textit{in data port} or a \textit{data access} feature, or even upon an error state change in the error model associated with the component under study or associated with a connected component.

The use of the Guard\_Out property resembles to the use of the Guard\_In property. For further details, please refer to [25].

\subsection*{2.1.6.2.5 Abstraction of error models}

Any component of the system model, the complete component hierarchy, can be associated with an error model. The error model of the root-level system component represents an abstract error model instance for the system. In general, an error model which is higher in the system hierarchy is an abstraction of the contained error models. AADL represents error models abstraction in two approaches: a basic error model that may be perceived as an abstraction of the behaviour of its subcomponents in presence of faults and a derived error model, whose error state is determined
in function of the error states of its subcomponents, and which may be viewed as an abstraction of those.

A basic error model describes the behaviour of a component independently from possible error states of its subcomponents, if any. This approach, of using a basic error model as an abstraction, can be used for a very detailed architecture model with its associated error models. Therefore, the error models associated with the subcomponents will be ignored by the analysis. Moreover, error propagations from subcomponent to external components will be abstracted as in and out, or an in out, propagations in the basic error model. This will alleviate the burden of carrying out the details of the subcomponents initiating or passing the propagation.

AADL offers the use of a property called Model_Hierarchy and which indicates whether the enclosing basic error model should be treated as abstract and thus, ignoring subcomponents’ error models in the analysis. The example below represents a basic error model as an abstraction of its subcomponents’ error models. Therefore, fault details of the subcomponents will be ignored by the analysis.

```plaintext
system computer
derived computer;

system implementation computer.personal
subcomponents
    CPU: processor Intel.DualCore;
    RAM: memory SDRAM;
    FSB: bus FrontSideBus;
annex Error_Model {**
    Model => My_ErrorModels::Example1.basic;
    Model_Hierarchy => abstract;
**}
end computer.personal;
```

The second approach for abstracting error models is the derived error model. In this approach, the error state of the derived model is determined by state mapping\(^\text{10}\), i.e. error states of subcomponents and error states and error propagations of connected components, and of connections as well, map to upper error states.

Abstractions based upon derived error models appear to be useful for some situations. Situations favouring, for instance, the restriction of the higher level view of the error behaviour to a smaller set of error states and outgoing propagations. As an example, a situation where it is required to partition the set of subcomponents error states based upon, for instance, catastrophic states [25]. Dependability evaluation tools may use such a partitioning.

The use of this approach consists of the use of three error model properties [25]:
- the typical Model property which associates an error model type to the component;
- a Derived_State_Mapping property that determines the upper level error states, and in function of error states and outgoing propagations of subcomponents and connections;
- error states and outgoing propagations of components located in the other end of an incoming feature, in port for instance;

---

\(^10\) Section 2.1.6.2.3. of this document - derived error model
• a Model_Hierarchy property value, which is assigned with the value derived.

2.1.6.3 Case study

The below is an example, aspired from the control process example presented in [22]. Alterations have been made to the example for simplification purposes and therefore, to cover the relevant issues of AADL from the perspective of dependability modelling. In that sense, error models have been made and have been associated with the component and also with its subcomponents. This is to examine some concepts such as abstractions of error models, state mappings to derived error states and error propagations studies.

2.1.6.3.1 an AADL component: structure, implementation and associated error model

The example in this section is about a process component which is identified as control.speed_control. The component type identifier is control, its implementation identifier is speed_control and its error model is the Example1 defined earlier in this document. The component has two input ports: an input data port, i.e. speed, and an input event port, i.e. brake, an output data port called throttle_cmd for a resulting command data.

Figure 19 - Graphical Representation of an AADL Sample Specification

The nominal structure of the component, text-represented, is as follows with control as the component type identifier, and speed_control as the component implementation identifier:

```plaintext
process control
features
speed: in data port raw_speed;
brake: in event port;
throttle_cmd: out data port command_data;
end control

process
implementation
control.speed_control
subcomponents
... connections
... end control.speed_control
```
The error model of Figure 15, i.e. Example1, may be applied to the control.speed_control implementation component and as follows:

```
process implementation control.speed_control

... annex Error_Model {**
    Model => My_ErrorModels::Example1.basic;
    Occurrence => fixed 0.4 applies to error CorruptedData;
**};
end control.speed_control;
```

The error model may be associated with any component that is an instance of the component implementation containing the "Model" property. However, the "applies to error" clause allows defining component specific occurrence properties for events and outgoing propagations, and which are declared in the error model type definition.

For the speed_control component, the probability of propagating data being corrupted is 0.4 rather than being 0.8 and as declared in the corresponding error model.

### 2.1.6.3.2 AADL subcomponents

At this stage of our study we examine, further in detail, the speed_control process which is an implementation component. Emphasis in this section is on its subcomponents, associated error models and error propagations between components.

**Connections, implementations and error models**

We consider two subcomponents: first, the input thread called GetSpeedEventData and which gets the speed data and the brake events. The second subcomponent is the output thread which controls the transferred data and then produces command data, it is called SetAndTransferCmd.

The graphical representation is therefore as follows:
The structural representations of the threads, i.e. software components, are given below and as text. To comply with the syntax\textsuperscript{11} of AADL, we use identifiers for component types, component implementations and component instances and as graphically summarised in Figure 14 of page 35.

The \textit{GetSpeedEventData} is an instance of the component implementation of the thread which responsible of getting the speed data and brake events. We assume that component has the name \textit{inputDataEvent} as its type identifier and the name \textit{inputDataEventProcessing} as its implementation identifier. The \textit{SetAndTransferCmd} is an instance of the component implementation of the thread responsible of controlling and then producing command data. We assume that component has the name \textit{outputCmd} as its type identifier and \textit{outputCmdProcessing} as its implementation identifier. The textual representations of subcomponents \textit{inputDataEvent} and \textit{outputCmd} are as follows:

\begin{verbatim}
thread inputDataEvent
features
  speed_data: in data port raw_speed;
  brake_event: in event port;
  proc_data_out: out data port processed_data;
end inputDataEvent

thread outputCmd
features
  proc_data_in: in data port processed_data;
  proc_cmd_out: out data port command_data;
end outputCmd
\end{verbatim}

\textsuperscript{11} Further details about AADL syntax are in (Feiler, P.H. et al. 2006).
And also the textual representation of the containing component as follows:\textsuperscript{12}

\textbf{process implementation} control.speed_control
\textbf{subcomponents}
GetSpeedEventData: \textbf{thread} inputDataEvent.inputDataEventProcessing
SetAndTransferCmd: \textbf{thread} outputCmd.outputCmdProcessing
\textbf{connections}
data port speed -> GetSpeedEventData.speed_data;
\textbf{event port} brake -> GetSpeedEventData.brake_event;
data port GetSpeedEventData.proc_data_out -> SetAndTransferCmd.proc_data_in;
data port SetAndTransferCmd.proc_cmd_out -> throttle_cmd;
\textbf{end} control.speed_control

For example the declaration

\begin{quote}
``data port GetSpeedEventData.proc_data_out -> SetAndTransferCmd.proc_data_in;``
\end{quote}
represents a connection, for processed data transfer, between the output data port \textit{proc.data.out} of the thread \textit{GetSpeedEventData} to the input data port \textit{proc.data.in} of the thread \textit{SetAndTransferCmd}.

We associate the error models Example1 and Example2 to subcomponents GetSpeedEventData and SetAndTransferCmd, respectively, and as follows:

\textbf{process implementation} control.speed_control
\textbf{subcomponents}
GetSpeedEventData: \textbf{thread} inputDataEvent.inputDataEventProcessing
SetAndTransferCmd: \textbf{thread} outputCmd.outputCmdProcessing
\textbf{connections}
data port speed -> GetSpeedEventData.speed_data;
\textbf{event port} brake -> GetSpeedEventData.brake_event;
data port GetSpeedEventData.proc_data_out -> SetAndTransferCmd.proc_data_in;
data port SetAndTransferCmd.proc_cmd_out -> throttle_cmd;
\textbf{annex} \textbf{Error.Model} {**
\textbf{Model =>} My_ErrorModels::Example1.basic \textbf{applies to} GetSpeedEventData;
\textbf{Occurrence =>} fixed 0.5 \textbf{applies to error} GetSpeedEventData.CorruptedData;
\textbf{Model =>} My_ErrorModels::Example2.basic \textbf{applies to} SetAndTransferCmd;
\textbf{Occurrence =>} fixed 0.5 \textbf{applies to error} SetAndTransferCmd.CorruptedData;
**}
\textbf{end} control.speed_control

The error model Example1 has a feature \textit{CorruptedData} which is an \textit{out error propagation}. However, the error model Example2 has a feature also called \textit{CorruptedData} but is an \textit{in out error propagation}.

\textsuperscript{12} The symbol “\textit{\rightarrow}” means an immediate connection between ports.
With this in mind, the *CorruptedData* is an *out error propagation* for the component GetSpeedEventData (because associated with Example1). It is, however, an *in out error propagation* for the component SetAndTransferCmd (because associated with Example2).

**Error propagation between components:**
Reference made to dependency and inheritance rules, and which have been summarised in sections 2.4.1 and 2.4.2, error propagations may occur between components GetSpeedEventData and SetAndTransferCmd. The associated error models Example1, which is associated with GetSpeedEventData, and Example2, which is associated with SetAndTransferCmd, perform the possible error propagations.

On the one hand, dependency rules state that an application component may propagate errors to every connection from any of its output ports, which applies to component GetSpeedEventData vis-à-vis its outgoing connection. Also, the dependency rule which states that a connection may propagate to every component that have an in port to which it connects, which applies to SetAndTransferCmd as a destination of possible error propagations from its incoming connection.

On the other hand, an inheritance rule also applies in this example. This is because we have not associated an error model with the connection between the components, i.e. GetSpeedEventData and SetAndTransferCmd. The “missing” error model is supposed to perform the error propagation through that connection. Therefore, we apply the inheritance rule stating that error propagation is passed from the component that is the origin of the connection\(^{13}\) to the component that is the destination.

Moreover, the component SetAndTransferCmd declares, in its associated error model, an incoming error propagation name that matches with the outgoing error propagation name of the error model associated with GetSpeedEventData. This is called a *name matching rule*\(^{14}\) for error propagations. We specify, e.g., Guard_In rules for the error model instance, i.e. Example2, and which is associated with the component SetAndTransferCmd. Therefore, we alter slightly the specification of the process implementation *control.speed_control*. Thereafter, it becomes as follows\(^{15}\):

```plaintext
process implementation control.speed_control
subcomponents
GetSpeedEventData: thread inputDataEvent.inputDataEventProcessing
SetAndTransferCmd: thread outputCmd.outputCmdProcessing
connections
data port speed -> GetSpeedEventData.speed_data;
event port brake -> GetSpeedEventData.brake_event;
data port GetSpeedEventData.proc_data_out -> SetAndTransferCmd.proc_data_in;
data port SetAndTtransferCmd.proc_cmd_out -> throttle_cmd;
annex Error_Model (**
Model => My_ErrorModels::Example1.basic applies to GetSpeedEventData;
Occurrence => fixed 0.5 applies to error GetSpeedEventData.CorruptedData;
Model => My_ErrorModels::Example2.basic applies to SetAndTtransferCmd;
Occurrence => fixed 0.5 applies to error SetAndTtransferCmd.CorruptedData;
```

\(^{13}\) It is the case of a connection without an associated error model to process the error propagation.

\(^{14}\) Exception made for a coming error propagation that is filtered by using a Guard_In property.

\(^{15}\) The added part is highlighted in italics.
Guard_In =>
   CorruptedData when GetSpeedEventData.proc_data_out[CorruptedData],
   mask when SetAndTransferCmd.proc_cmd_out[Error_State_Failed]
applies to SetAndTransferCmd.proc_data_in;
**};
end control.speed_control

The added part (in italics) are of the specification, and means that there is an unconditional
mapping of the outgoing error propagation, out CorruptedData, declared in the error model
instance associated with GetSpeedEventData. The propagation occurs through the feature
proc_data_out of the component. This unconditionally maps to the in out error propagation, named
Corrupted Data, and declared in the error model instance associated with the component
SetAndTransferCmd.

The second rule specified in the Guard_In property masks the incoming error propagation, and
which was processed by the error model instance of SetAndTransferCmd. It will be masked at the
condition of the error model instance of the component feature proc_cmd_out is in the error state
cited above (Error_State_Failed). If the mask condition evaluates to true then, the incoming error
propagation, and which has been mapped and named unconditionally in the preceding rule, will be
suppressed. Therefore, this study shows that, and with the existence of the mask specification,
any propagation of error, even though unconditional, may be cancelled upon the evaluation to true
of some conditions.

2.1.6.3.3 Abstraction of AADL subcomponents

Previously, two abstraction approaches have been presented: the basic error model and the
derived error model abstraction. As an example, we will apply the basic error model approach to
abstract both subcomponents’ error models, i.e. error model instances associated with
GetSpeedEventData and SetAndTransferCmd, respectively.

The declaration below specifies the error model, and which is associated with the speed_control
implementation of the process control, as an abstract error model. We assume the associated
error model has the name Example3. The italicised section declares the error model Example3 as
an abstract model for the speed_control. Therefore, the underlined section, since it declares fault
details of subcomponents, will be ignored by the analysis and thus could be removed. This
includes the specified rules for managing propagations\(^\text{16}\).

\begin{verbatim}
process implementation control.speed_control
subcomponents
GetSpeedEventData: thread inputDataEvent.inputDataEventProcessing
SetAndTransferCmd: thread outputCmd.outputCmdProcessing
connections
data port speed -> GetSpeedEventData.speed_data;
event port brake -> GetSpeedEventData.brake_event;
data port GetSpeedEventData.proc_data_out -> SetAndTransferCmd.proc_data_in;
data port SetAndTransferCmd.proc_cmd_out -> throttle_cmd;
annex Error_Model {**
Model => My_ErrorModels::Example3.basic;
Model_Hierarchy => abstract;
****}
\end{verbatim}

\(^{16}\) They are propagations incoming to (and outgoing) from subcomponents
Model => My_ErrorModels::Example1.basic applies to GetSpeedEventData;
  Occurrence => fixed 0.5 applies to error GetSpeedEventData.CorruptedData;
Model => My_ErrorModels::Example2.basic applies to SetAndTransferCmd;
  Occurrence => fixed 0.5 applies to error SetAndTransferCmd.CorruptedData;
Guard_In =>
    CorruptedData when GetSpeedEventData.proc_data_out[CorruptedData],
    mask when SetAndTransferCmd.proc_cmd_out[Error State Failed]
applies to SetAndTransferCmd.proc_data_in;
**);
end control.speed_control

Yet incoming propagations to subcomponents from external ones as well as outgoing errors from subcomponents to external components will be abstracted in the upper level error model. Therefore, the incoming propagation to a subcomponent from outside will appear an incoming propagation to the upper component. Similarly, outgoing propagation from a subcomponent to outside will appear as outgoing propagation from the upper level component to outside. Furthermore, the first rule following the Guard_In property will no longer be meaningful in the upper level model. It is a rule to manage propagation between two subcomponents; unconditional even so. However, we still need to express that rule which masks the error propagation and thus prevents it from propagating to outside.

A suggestion is such that the proc_cmd_out feature of the subcomponent SetAndTransferCmd needs to be propagation analysed in the upper level model and also needs to be analysed for failures, in which case the propagation will be masked. Another issue is the applies to clause which need to apply to a feature, i.e. an in port, of the upper level model. A suggestion is such that a subcomponent in port connected to a subcomponent out port will be abstracted as the other end of the in port of the first connected subcomponent, but the in port which is fed from outside. In our example, it is the speed port.

2.1.6.4 Discussion

AADL allows the use of operational modes and mode transitions to describe systems which are adaptable to escalating development costs, reliability and performance requirements [25]. Although AADL does not describe how implementation details are to be specified, the relevant implementation characteristics are described as component properties and as rules of conformance between the properties and the described components.

One deficiency of the language consists of the incomplete support, at least in its core concepts, of analysis of the runtime architectures. This shortage is compensated by extensions to accommodate analysis specific notations that can be associated with components. Error modelling for instance is supported through an annex that has been added to the standard. Nevertheless, AADL support of both software and hardware architectures can be used to separate software faults and hardware faults.

In the context of dependability, safety requirements can be specified by means of properties associated with each component, failure modes of components can be specified in the added error model annex. This annex uses a state-based formalism to define error models [25]. Each AADL error model can be stored in a library and can be used to annotate AADL components [22]. It is possible to adapt and overwrite certain characteristic, such as the occurrence probabilities of failure transitions, of the error models when it is used for a specific component.
Furthermore, the language specifies error propagations in the context of compliance with rules. Those rules may be issued from components’ dependencies or from inheritance. The latter means relations in both directions from child component to parent or from parent component to a contained one, even though not direct, i.e. to a child. To summarise the purpose of inheritance rules, it is to find a ‘relative’ component, i.e. child or parent, but with an associated error model instance. That instance will process the error propagation, and as a substitute of that one missing in the involved component.

Always in the context of error propagations, AADL offers the capability to map unconditionally (and by propagation name matching) outgoing propagations to incoming ones. Moreover, the language specifies a mask rule which has the capability to suppress error propagations, even though unconditionally mapped, if some conditions evaluate to true. With this in mind, it may be useful to know a potential capability of masking a set of propagations, and uniformly. This may help in isolating one or more propagations for safety assessments, and possibly automated.
2.2 Behavioural Fault Simulation approaches

In behavioural fault simulation, system failure models equivalent to an FMEA are produced by injecting faults into executable formal specifications of a system, thereby establishing the system-level effects of faults. In most cases, the results are then used in a model-checker to verify the system safety properties. Techniques that follow this approach include safety analysis using Altarica, FSAP-NuSMV, Deviation Analysis, and DCCA.

2.2.1 Altarica

Altarica [26, 27] is a language designed to be able to formally describe complex systems. As with compositional techniques, Altarica allows systems to be represented as hierarchies of components and subcomponents, and as with the State-Event Fault Trees described earlier, it models both events and states.

Components in Altarica possess a number of state and flow variables, representing the internal state and the external interactions of the component respectively. For example, a simple electrical switch would have one state variable (on/off) and two flow variables representing input and output voltage. The sets of state and flow variables are always discrete, however, meaning that the configuration of a component can only change by increments. Thus the voltage flow variables may be treated as containing only two discrete values, 'high voltage' and 'low voltage'. Whenever a variable changes from one value to the next, this is known as a transition and must be caused by some event.

There are two types of events in Altarica: local events, so named because they are known to the component in question, i.e. they are either internal or immediately connected to its interface in some way; and invisible events, which are external events not visible to the component in question. An example of the former event type would be the flipping of the switch – an event which is perceived by the component. An example of the latter would be the failure of the power supply feeding voltage to the switch – the switch would not know of the power failure directly, yet its input and output voltages would both simply change to the 'low voltage' value. The distinction is based on the function of the component, e.g. if the switch had a voltmeter sensor attached, it would be able to detect the drop in voltage to its input and would be aware of the power failure event. Events can also be assigned priorities, so that an event can occur only if there are no higher priority events.

Each basic component is therefore described by an interfaced transition system, consisting of a description of the possible events, possible observations, possible configurations (i.e. combinations of state and flow variables), mappings of what observations are linked to which configurations, and what transitions are possible for each configuration. These basic components can be composed into nodes, which are sets of subcomponents or other nodes acting together under a controller (an entity responsible for coordinating the subcomponents). The top-level node represents the system itself. Like normal components, nodes are described by an interfaced transition system, but in addition to consisting of events, observations, and optional subnodes/components, a node also has a description of its controller and a set of broadcast synchronisation vectors.

These broadcast synchronisation vectors allow events in one node or component to be synchronised with those in another. These vectors can contain question marks to indicate that an event is not obligatory (e.g. a bulb cannot turn off in response to a power cut if it is already off). Additional constraints can be applied to the vectors to indicate that certain combinations or numbers of events must occur, particularly in the case of these 'optional' events, e.g. that at least one of a number of optional events must occur, or that k-out-of-n must occur etc.
The controller is responsible for coordinating the flow variables of the subcomponents. Its configuration allows certain relationships between the various subcomponents, e.g. ensuring that the relationship between a pump and a valve means that flow is only possible when both the pump is operating and the valve is open. Similarly, any transition that occurs in the controller may also effect a transition in the subcomponents, or vice versa, and thus a relationship is established between the behaviour of the subcomponents in a node and the behaviour of the node itself (as represented by the controller). Controllers play a particularly important role in a node because they are also capable of reconfiguring the system in response to failures. For example, if a subnode failed according to event ‘failure’, which moves the state of the subnode from 'OK' to 'Failed', then the controller may activate a standby subnode instead. Note that because state variables are internal to a component (or node), a synchronised event 'subnode_fails' is necessary to link the failure of the subnode with the transition in the controller.

An example description of a simple, single-state node is given below:

```
node block
  flow
    O : bool : out ;
    I, A : bool : in ;
  state
    S : bool ;
  event
    failure ;
  trans
    S \= failure \rightarrow S := false ;
  assert
    O = (I and A and S) ;
  extern initial_state = S = true ;
``` 

Here, the node 'block' contains three flow variables (O, I, A) and one state variable (S). There is one event, failure, that causes the state to transition to false, i.e. the component becomes inoperative when it fails. The assertion links the flow variables such that output only occurs when input is present, an active signal is present, and the component is functioning (i.e. S = true, which is the initial state).

Once a system has been modelled in Altarica, it can be analysed by external tools and methods, e.g. the generation of fault trees, Petri nets, model-checking etc. Each has specific roles in the assessment of an Altarica model, so fault trees could be used for non-dynamic failures, and model-checking could be used for verification purposes. Construction of fault trees, Petri nets, and model-checking input (e.g. in the SMV language) is automatically carried out by the relevant external tools and produce a number of different types of results. For example, Altarica can be converted into fault trees which can then be analysed to produce minimal cut sets or prime implicants (which are like minimal cut sets except that they can also contain negations, e.g. A.¬B would mean that A must occur but B must not in order to cause system failure). Model-checking can determine whether or not the failure combinations calculated by the fault trees can be eliminated by modifications to the controllers so that those combinations should not be able to
occur. Thus both the static and dynamic elements of the model can be analysed, albeit by different tools, to ensure that the system meets its safety requirements.

There are a number of drawbacks to Altarica, however, when applied to real-world systems [28]. One particular issue is the difficulty of modelling the propagation of failure through the system; for example, to model a loss of pressure in a hydraulic system, the 'fluid' and 'pressure' information must both be transmitted around the system, via bidirectional signals. This bidirectionality, particularly in electrical or hydraulic systems, can frequently lead to loops or circular propagations in the model, which will be rejected by Altarica. Although this can be solved by adding delays, the delays can mean that instantaneous propagations are not modelled correctly.

Another issue is that of fault tree generation; as mentioned above, Altarica models can be used to generate fault trees, but only in cases where the dynamic or temporal ordering of events is not an issue. Even systems which are not dynamic but which have had to be modelled with delays to overcome circular propagations will suffer from this problem. Instead, other methods must be used to analyse such systems (e.g. model-checking), often much less efficiently. Instead, Altarica's main strengths lie in its formal language and the ease with which this can be converted into formats suitable for model-checking and similar formal analysis techniques.

### 2.2.2 FSAP/NuSMV

The Formal Safety Analysis Platform FSAP/NuSMV-SA [BOZZANO03, 29, 30] actually consists of two parts – a graphical user interface and tool (FSAP) and a model checking engine (NuSMV) with safety assessment capabilities (-SA). It is intended to provide a uniform environment for the modelling and assessment of safety-critical systems. Analysis is possible in both nominal and degraded modes, allowing the user to inject particular failure modes into the system and observe the effects. It also offers typical model-checking capabilities, e.g. property verification and the ability to generate counter-examples, and there are algorithms that allow NuSMV models to be used as the basis of other types of analysis, e.g. by automatic conversion into fault trees.

FSAP/NuSMV-SA is primarily designed for checking that a system model can meet its safety requirements, but it is also capable of performing safety analyses itself. It provides a platform for the whole process, beginning with the annotation of the model with failure modes, the definition of safety requirements, simulation of injected faults, and finally verification of the system safety requirements. As mentioned, FSAP/NuSMV-SA is capable of fault tree analysis, including NOT gates (i.e. specifying that an event must not occur in order to cause a system failure), and is capable of a failure ordering analysis in cases where the order of events is important (in which case, standard FTA is insufficient).

The chief benefit is the ability to use the same system model for multiple purposes: the design models can be used for safety analysis and verification purposes. This means that safety analysis is more tightly integrated with the design and will evolve at the same pace; the shared model approach also makes it easier to use in an incremental design process, refining the model by adding new failure modes and/or safety requirements as needed.

The joint system design/safety assessment process used with FSAP/NuSMV-SA is generally as follows:

- A formal model is developed with NuSMV. This can be a nominal system model or a formal safety model.
- The next step involves injecting failure modes into the model (if not already present) to produce an extended system model, which describes the failure modes by which the components can fail. These failure mode annotations can be stored in and retrieved from a library for a greater degree of reusability.
• The model is then augmented with safety requirements. Again, these can be stored in and retrieved from a library.

• Next, the behaviour of the annotated model is assessed against its functional and safety requirements. This involves analysing the model, using the model checker to verify that system properties are valid and generating safety analysis results by means of fault trees etc.

• Finally, the results of the analyses are processed and presented to the user. These results can then be used as the basis for decisions about how best to improve the system design model.

The annotations for the model are written in the NuSMV textual specification language, designed to represent finite state machines. It provides for hierarchical descriptions as well as reusable components and generally takes the form of a set of temporal logic expressions. These expressions can then be parsed by symbolic algorithms and checked to ensure that the specifications hold. An example NuSMV model for a two-bit adder is shown below:

```nuSMV
MODULE bit(input)
VAR
    output : {0,1};
ASSIGN
    output := input;

MODULE adder(bit1, bit2)
VAR
    output : {0,1};
ASSIGN
    output := (bit1 + bit2) mod 2;

MODULE main
VAR
    random1 : {0,1};
    random2 : {0,1};
    bit1 : bit(random1);
    bit2 : bit(random2);
    adder : adder(bit1.output, bit2.output);

This can then have a fault injected into the bit module as follows:

VAR
    output_nominal : {0,1};
    output_FailureMode : {no_failure, inverted};
ASSIGN
    output_nominal := input;
DEFINE
    output_inverted := !output_nominal;
    output := case
```

© 2010 The ATESSST2 Consortium
output_FailureMode = no_failure : output_nominal
output_FailureMode = inverted : output_inverted

ASSIGN next(output_FailureMode) := case
                  output_FailureMode = no_failure : {no_failure, inverted};
                  output_FailureMode = inverted : inverted;
                  esac

This is injected by means of the FSAP interface rather than being typed in manually (it can even be chosen from a library) and replaces the original definition of output, taking into account the possibility of a corrupt input bit by defining an output_FailureMode variable. If this variable is false (i.e. no_failure) then the output is the nominal output, i.e. the same as the input, and if it is true (i.e. inverted) then the output is the negated value instead.

Annotations like these will later be analysed in one of many ways. They can be used as the basis for a simulation of the model, the output of which is a trace of the system failure; they can be used as part of a property verification process, either proving that a property holds or producing a counter-example demonstrating why it does not; they can be used to construct fault trees automatically, to calculate the root causes of a given failure event; or they can be used as part of a failure order analysis.

It is worth delving into these last two a little further. FSAP/NuSMV-SA produces fault trees by first performing a reachability analysis, beginning with the initial states of the system and iteratively producing a set of successor states until a state in which the top-level failure event of the fault tree occurs is reached. The result is the set of all states in which that failure event occurs. All other system variables are then removed from these states, leaving only the failure variables (i.e. the occurrence or not of the various failure modes), and then finally the resulting expression, consisting of a set of occurring and non-occurring failure modes, is simplified to obtain the minimal cut sets for the fault tree. This process ignores the dynamic information stored in the model (since classical fault trees are a purely combinatorial structure), so to ensure that the dynamic behaviour of the system is not lost in the analysis, NuSMV-SA can associate traces with each cut set to show how the failure can occur.

The other way of analysing the dynamic information is to perform a failure order analysis instead. This is based on the FTA process but includes the timing and ordering constraints that apply to the cut sets. The two processes can be run together, i.e. the basic information required by the failure order analysis is also required by the FTA, with the addition that an ordering information model is also produced, which forms the basis of the failure order analysis. The outcome is a precedence graph showing the order of any events which must occur to cause the top failure event of the fault tree.

Although FSAP/NuSMV-SA has a number of powerful tools at its disposal, it does also have a number of drawbacks, particularly in regard to its handling of fault trees. Firstly, the trees produced are flat (and not very tree-like) in that they show direct links between the top event and the root causes; there is no indication of the propagation through the system, and this can have readability implications for complex systems. Secondly, the handling of common cause and transient failures is limited. Thirdly, only qualitative FTA is possible, i.e. there is no ability to perform a probabilistic evaluation of the fault trees (particularly in terms of the failure order analysis). Finally, and perhaps most critically, FSAP/NuSMV-SA – like other formal, model-checking based approaches – suffers from state space explosion in large or complex models, though models can be simplified in various ways to help counteract this problem.
2.2.3 Model-based Deviation Analysis

Deviation Analysis is the study of how inaccuracies – or deviations – in the measurements of monitored variables will affect the behaviour of a system. This differs from other safety analysis such as those above in that it does not aim to discover or analyse faults; instead, it assumes that the system itself operates correctly and is simply being fed incorrect input data. This type of deviation analysis originated with Software Deviation Analysis or SDA [31], which allowed exploratory analysis of deviations on system inputs specified with qualitative descriptions (i.e. categories rather than actual values, thus ‘high’, ‘low’ etc). SDA allows an analyst to answer questions like, “What would be the possible effects of an altimeter deviated so that it always reads higher than the correct value?” The process of SDA involved taking a blackbox software or system specification and providing information about particular input deviations and hazardous outputs; SDA would then automatically produce scenarios in which the input deviations could lead to the hazardous outputs. SDA itself is based on the HAZOP technique, which considers possible deviations for a set of system variables (e.g. ‘too high’ for the variable ‘tank pressure’).

Whereas SDA was an exploratory, qualitative technique, model-based deviation analysis uses model checking to perform verification on the system [32]. As such, it is more precise and can use quantitative values to determine whether the deviations will invalidate the specification, e.g. "Will an altimeter that deviates by 0 to 100 feet too high have an effect on the autopilot?" In a general sense, the model-checking approach to deviation analysis is designed to answer questions of the form: "Given a deviation of \( x \) and \( y \), will the resulting deviation of \( z \) still be within an acceptable range?" rather than the basic ‘what if’ questions posed by standard SDA.

Model-based deviation analysis works by comparing the outputs of two almost identical models, one that uses only the nominal behaviour and another that includes the input deviations. Any differences in output are therefore due only to the differences in input. A model checker (e.g. NuSMV) then determines whether these output deviations cause system specification properties to be broken. In common with most model-checking approaches, however, deviation analysis suffers from state-space explosion, particularly since it effectively analyses two models, potentially doubling the number of variables involved. The fact that the variables often have large domains (rather than simple qualitative categories) also adds to this problem. This can be alleviated to an extent by analysing both models (nominal and deviated) simultaneously by embedded them in a common framework, so that they share common variables.

2.2.4 DCCA

DCCA, or Deductive Cause Consequence Analysis [33, 34], is a formal safety analysis technique that uses mathematical methods to determine whether a given component fault is the cause of a system failure or not; in other words, it answers the question, "What combination of failures can lead to a given system hazard?" It combines elements of both FTA and FMEA but is more expressive than FMEA and more formal than FTA.

In DCCA, system models are represented using finite automata with semantics represented in a temporal logic known as CTL (Computational Tree Logic). Then, for each set of component failure modes (represented as logical predicates), a property is defined in CTL that specifies whether or not that collection of component failure modes can cause a given system failure. This is known as the criticality property and can be extracted automatically from the finite automata in the system model; if it is true, then the set of failure modes is a critical set, analogous to a cut set in normal fault trees. The goal of DCCA is to discover all minimal critical sets, i.e. sets of failure modes that are sufficient and necessary to cause a given system failure.

Clearly, to automatically calculate and check each possible set of failure modes would require exponential effort, so DCCA is designed to make use of initial results from more informal methods (such as FTA) and then verify this smaller number of potential results. It can also make use of results from formalised versions of FTA, such as Formal FTA, which annotates each gate with
CTL semantics and which enforces additional properties (such as completeness) upon the fault tree. However, Formal FTA requires that all nodes be formalised, i.e. annotated with CTL, whereas DCCA does not require formalisation of intermediate nodes / logic gates; this both reduces the amount of time required to create the fault trees and also avoids problems where inner nodes are too vague to be easily formalised. Although these various techniques help to reduce the impact of the state-space explosion problem on DCCA, like all formal model-checking approaches, it is prone to performance and efficiency problems in certain (albeit worst case) scenarios.

Furthermore, as already established, one of the main problems with fault trees is their inability to model dynamic situations where the order of events is important. A more recent extension to DCCA, known as Deductive Failure Order Analysis [35], enables DCCA to take these situations into account as well. In this extension, temporal fault trees containing Pandora-style PAND and SAND gates are automatically synthesised from a DCCA -annotated system model. DCCA is then applied to obtain the unordered minimal critical sets and then DFOA is applied to restore a partial ordering to these using semantics based on CTL* (an extended version of CTL with additional operators). The results then specify either a basic combination of failure modes, a sequence of failure modes (using PAND), or a set of simultaneous events (using SAND) that must occur to cause the system failure.

2.2.5 Automatic FMEA Approaches

Most of the approaches so far have been based on or have made use of fault trees. There are some approaches that use FMEA as the engine for safety analysis instead of fault trees. One such technique is AutoSteve [36], which is capable of looking at the effects of multiple failure modes, particularly designed for electrical and electronic systems.

AutoSteve functions by means of qualitative simulation of a nominal system model and a degraded system model and compares the results, similar to Deviation Analysis; the results are abstracted and then the effects of the failures (i.e. the differences from normal behaviour) are produced. Qualitative simulation has a number of advantages over quantitative simulation; firstly, detailed data about the parameters of components are unnecessary, something which is particularly valuable at early stages in the design process when the exact implementation details are still unknown. It is also much more efficient, which is important given the cost of performing an inductive multiple failure FMEA. Finally, qualitative descriptions are very reusable, as the description for a generic switch can be used for many actual implementations of a switch, even if they would have different quantitative parameters.

Three different types of information about component behaviour is needed by AutoSteve: the interface, i.e. inputs and outputs of the component; the internal topology, making up the links and relationships between inputs and outputs; and any internal dependencies, i.e. how one part of the component reacts to changes in another part of the component. This information can be represented by dependency expressions or by state charts and would be added to a diagram of the system, e.g. a circuit diagram in a CAD tool. The resulting model is then used as the basis of the simulation: beginning with the initial states of all components, the system is simulated until no more state changes take place. If failures are injected into this simulation, then it allows the behaviour to be compared with the nominal (non-faulty) behaviour. The results are ultimately presented in the form of an FMEA, i.e. for each component failure, it describes the effects on the rest of the system.

Multiple failures can be analysed by positing pairs, trios, and so forth of failures in a single simulation. This leads to a combinatorial explosion in the number of possibilities, which has negative implications for both performance as well as for the analyst or engineer who has to then absorb and understand the results. The number of possibilities can be reduced to an extent by eliminating mutually exclusive combinations of failures, particularly on the same component (e.g. a switch being both stuck shut and stuck open at the same time). AutoSteve further reduces the
scale of the problem by focusing only on the most significant failures, e.g. by introducing a
probability threshold (very unlikely faults are ignored) and similarly for combinations of failures
(very unlikely combinations are ignored). The possible combinations are calculated iteratively, e.g.
first all single failures, then all pairs involving those single failures, then all trios involving the likely
pairs and so forth. Causal relationships between multiple failures, that is a combination of failures
in which one fault causes another, need to be handled differently in order not to affect the
quantitative results.

Pruning of faults can be problematic when unlikely faults are also the more severe; a combination
of failures can at times be more severe than a single (often more likely) failure, e.g. the loss of all
windscreen wiper functions rather than the loss of just, say, the intermittent wiper function.
Furthermore, pruning in this way is only possible when such probabilities are available and to an
extent counters the main advantage of qualitative simulation, namely the fact that quantitative data
(such as probabilities) does not need to be incorporated into the model. In some cases
probabilities might not even be available, though they can often be approximated, and since the
probabilities are used only for pruning, this should not affect the results unduly.

AutoSteve can also be used to a limited extent for fault diagnosis. By grouping all causes of a
given fault together, a rudimentary diagnostic tree can be constructed. This can then be used to
determine the cause of a given failure by comparing observed symptoms against the component
failures given in the tree. However, in this case, pruning may also eliminate the correct diagnosis
which means more caution is needed when reducing the number of results.

A similar tool to AutoSteve is Ricardo's AutoFMEA tool, which is integrated into Matlab Simulink
[37]. Like AutoSteve, it is an automated FMEA tool that determines the effects of single or multiple
failures by injecting them into a simulation of the model. Components in the model are annotated
with possible faults, which can be stored in a library. Faults have a type (e.g. delay, cut off, etc), a
failure rate, and any necessary external parameters (e.g. for a delay, the time interval), and these
are then associated with outputs from components in the model. Hazards are defined for the
system and can be caused by one or more faults occurring in combination; the simulation then
attempts to determine whether the injected faults are capable of causing any of the defined
hazards, beginning with the initial system state (which may be degraded if a fault has been
injected) and any specific input values and iterating until either a hazardous state has been
reached or it is determined that no such path exists, in which case the fault does not cause any
hazards. The different possible starting states (including the nominal, non-degraded simulation)
are known as scenarios.

The results are generated in the form of an FMEA table indicating which faults cause which
hazards together with severity, probability, detectability, and occurrence values (from which an
RPN can be calculated). As with AutoSteve, the results can also be compared against the nominal
behaviour of the model, and to help manage the number of results, failures can be mapped to a
simple 1 to 10 scale. This makes mapping the failures to an FMEA simpler and also provides a
means of ranking the results so that the analyst is not overwhelmed by the amount of data. To
help reduce the number of results and also improve performance, simulation can be restricted to
certain parts of the system or only faults with a certain level of probability.
3 Online Fault Monitoring & Diagnosis

This section describes a number of different methods and techniques for the diagnosis of faults. Unlike the previous section, these techniques are intended to be used on operational systems rather than predicting or assessing potential faults in a system design. As a result, they tend to make heavy use of monitoring and system observations to provide the data necessary to determine the cause of the fault.

3.1 Introduction

It is one thing to attempt to anticipate and correct potential faults in a system design, but quite another set of techniques are required to diagnose problems in existing, operational systems. These techniques generally begin with observing certain system parameters and detecting when those parameters become abnormal. The challenge is in relating those abnormal measurements (known as ‘symptoms’) to a probable cause – i.e. diagnosing the root of the problem – and thus allowing remedial action to take place. The complication in this process comes from the fact that faults normally do not have single, mutually exclusive effects, and thus it is typically not possible to determine a single responsible fault from a single given effect; instead, complex failures often manifest multiple similar symptoms, making the diagnosis of the original fault much more difficult and requiring more sophisticated methods and a greater degree of expertise to achieve.

It is for this reason that many different diagnosis techniques have been developed over time. The complexity of the task in particular has led to a focus on automated techniques that are capable of diagnosing a problem with minimal manual input. This requires some level of knowledge about the system on the part of the diagnosis method as well as some means of determining the current state of the system operation; in many cases, this means predicting how the system should be operating and comparing this prediction against how the system actually is operating.

The various diagnosis techniques can be divided generally into three basic classifications, depending on how the knowledge about the system is coded or stored. The first category is rule-based diagnosis, which represents the knowledge about the system in the form of a set of if-then rules linking cause to effect – “if A occurs then B will result”. These rules can be used for diagnosis both forwards (“Has A happened? If so, what might happen next?”) and backwards (“B has happened – what occurred to cause it?”) when a given rule is ‘triggered’, i.e. a particular condition is met.

The second category is the model-based approach. In this approach, knowledge about the system is stored within a model of the system (in some cases a model that can also be used for other purposes and may indeed be a derivative of the design model). This model can either represent the normal functioning of the system, in which case diagnosis takes place when the actual behaviour of the system deviates from the normal behaviour predicted by the model, or the model can represent the failure behaviour of the system, modelling the effects and causes of different types of faults. The former type of model can be a model of the functional decomposition of the system, a model of the system architecture, or a behavioural simulation model, depending on the goal; the latter type of model typically tends to model either faults and their propagation through the system or causal processes describing the relationships between different processes and variables and the effect they have on each other.

The third category is the data-driven approach, which is based on the monitoring of the system and comparison against historical trends in these observations. In this type of approach, diagnosis takes place when the observations deviate from expected trends.
3.2 Rule-based expert systems

In this first type of approach, diagnosis is performed by detecting any anomalies in the system operation and then attempting to determine the causes of those anomalies by means of rules that describe the relationships between faults and their effects on the system. Thus if anomaly A occurs and we know that fault F is the only fault that causes A to appear, then we can assume that fault F has occurred. This approach has its roots in medical diagnosis, e.g. the MYCIN expert system [39], and was later applied to engineering systems in the 1980s. Expert systems feature a knowledge base that contains both declarative knowledge ('facts') – statements that describe the system, e.g. its initial configuration or state – and procedural knowledge ('production rules') that describes the functional relationships between system components and causal relationships between failures and their effects. The rules are generally formed either by system experts or derived from models and theoretical studies of the system.

Diagnosis in rule-based systems can be achieved both with a forward and backward chaining of the rules [40,51]. Forward chaining entails monitoring certain system parameters and activating a particular rule once a trigger condition has been met. The 'consequent' of the rule (i.e. its effect) is then considered a new fact about the system and may in turn trigger one or more new rules. If the consequent of the rule is a system failure, then this chain of activated rules yields the sequence of events that caused it. Alternatively, backward chaining can be used; this is a different inference mechanism that involves testing hypotheses [42]. A hypothesis is usually a malfunction that is assumed to have occurred on the basis of some abnormal system activity. Testing this hypothesis is accomplished by checking to see if the hypothesis is the consequent of any rules in the knowledge base; the 'antecedents' (i.e. prerequisites) are then taken as a new set of hypotheses to be tested for. This process continues until either the hypothesis is proved false (because some antecedents are not true) or the hypothesis is verified (because all the antecedents can be confirmed to be true by means of real-time monitoring or observations). Both forward and backward chaining can be used together, e.g. in REACTOR [43,54], a nuclear power plant monitor.

One problem with many early rule-based expert systems was an inability to remember events and thus an inability to perform any temporal reasoning. This made it difficult to diagnose many problems where both the short-term and long-term trends were important and where the sequence of events was critical. More recent rule-based systems feature temporal extensions to overcome this deficiency [45]; for example, RT-WORKS is a rule-based expert system which can monitor system parameters and also keep associated time values, thus allowing diagnoses based on the history of the system operation as well as its current state [46]. RESHELL is a similar example that can perform diagnosis of gradually deteriorating engineering systems [47].

Temporal reasoning is not the only difficulty with rule-based diagnosis, however; a more fundamental problem is the uncertainty inherent in the relationships between observations about the system and the conclusions of the diagnosis. Rules can only be confirmed by measuring system parameters, and those measurements do not necessarily have an absolute level of confidence or accuracy. Sometimes the rules themselves are not certain, e.g. if there several possible antecedents that cannot be checked for. In such situations, the most common solution is to use 'certainty factors', which are heuristic numbers that associate a certain level of confidence with each rule. Certainty factors allow diagnoses to be reached with different degrees of certainty. MYCIN originated this solution but it has been applied in other systems since, e.g. the mineral exploration expert system PROSPECTOR [48].

Rule-based expert systems also offer a number of benefits, however, particularly with regard to the separation of knowledge from reasoning. This separation makes them relatively independent of application, as the principles involved can be applied to many different domains, from medicine to engineering to mining. The ability to use both forward and backward chaining is also a benefit as diagnoses may be reached (or confirmed) using more than one method. Furthermore, the flexible nature of rule-based systems has led to them being combined with neural networks to
enable the detection and diagnosis of symptoms of failure in industrial processes; e.g. QUINCE [49,63] is a hybrid system that uses neural networks to detect symptoms and a rule-based system to relate those symptoms to their root faults. If the neural network is trained on the normal behaviour of the system, then the prediction error of the network indicates when the behaviour unexpectedly deviates from normal, indicating anomalies in the system that can be checked against the rules.

Generally speaking, rule-based expert systems have been most successful in processes that can be described with a small set of rules. Attempts at creating larger rule-based diagnosis systems for more complex systems have shown that such diagnoses are prone to inconsistencies, incompleteness, long search times, and a lack of maintainability [51,64]. Many of these problems are a consequence of the limitations of the production rule as a method for capturing knowledge; although they are effective at capturing and extending knowledge in a consistent fashion, the simple implication relationships provide little flexibility for analysts trying to represent the failure behaviour of a system. As a result, more elaborate methods for representing knowledge were developed, leading to model-based diagnostic systems.

### 3.3 Model-based approaches

Model-based approaches rely on expert knowledge stored within a separate model of the system being monitored. This provides an additional dimension to the diagnosis by making it possible to better describe and understand the relationship between symptoms and causes, allowing a deeper diagnosis to be made than by a classification-based diagnosis systems based only on rules; this is because knowledge about the system is represented by the system model itself instead of storing knowledge in a database about the system. Thus diagnosis is made not on the basis of trying to match patterns against a set of stored knowledge, but via an understanding of the causal processes in the system that lead from cause to symptom [53].

There are two types of model – those representing the normal behaviour of the system and those representing the abnormal behaviour of the system. Abnormal behaviour models allow diagnosis on the basis of determining the occurrence and causes of faults and include models of fault propagation and causal processes, whereas normal behaviour models include functional models, behavioural models, and qualitative simulation models. In these cases, diagnosis normally takes place by comparing actual system behaviour against the normal behaviour predicted by the models. Both types are described below.

#### 3.3.1 Fault propagation models

One of the earliest type of fault propagation models to be used in monitoring & diagnosis systems were the Cause Consequence Diagrams (CCDs), used in STAR [54], a small experimental system for the OECD Halden Reactor in Norway. CCDs represent the causal relationships between the status different components of a system in a logical, formal way by means of a network of cause and consequence trees. Cause trees describe how combinations of system malfunctions can lead to a hazardous event while consequence trees start with the assumption of a hazardous event and then describe how other conditions and the success or failure of mitigating actions can lead to safe or unsafe states in the system.

In STAR, CCDs formed a monitoring model which was periodically updated with readings from the system; the model was then repeatedly examined for the occurrence of chains of events, and if detected, these chains were then extended in both directions to determine the possible causes and also the possible consequences. A similar method is used in EPRI-DAS [55], which uses a Cause Consequence Tree (CCT) – a modified cause tree extended to include delays and timings.
to detect patterns of events that could indicate the early stages of faults propagating through the system.

Figure 21 - Cause Consequence Diagram

Although CCDs suffer from problems of computational efficiency [56, the principles involved in early CCD-style model-based systems have endured and also appear in newer safety monitoring systems, e.g. for the Surry & San Onofre nuclear power plants in the USA [57], which use master fault trees [58] synthesised from a PRA as the fault propagation models to assist real-time safety monitoring. An alternative method that instead uses fault trees for diagnosis is a hybrid system [59] in which neural networks perform the initial detection of symptoms and a rule-based system performs the diagnosis using knowledge derived from fault trees produced during a PRA. The fault trees are converted directly into the production rules that describe the relationships between each level of the fault tree.

Fault trees can be used directly as a diagnostics tool, typically in a two part process that uses fault trees together with a network of system sensors to enable the detection and diagnosis of failure effects using a simple pattern recognition process [60]. In this case, the fault trees are generated offline by gathering information about possible causes and consequences of failures in the system. Included in this process is the identification of system variables that need to be monitored by additional sensors. Once the fault trees are complete, all the different failure scenarios are considered and a pattern of behaviour as would be detected by the sensors is associated with each one. When the system is online, the behaviour as reported by the sensors is compared against the patterns generated earlier, and if a match is found, then a diagnostic fault tree is prepared that displays all possible scenarios that could cause that pattern; this tree includes time-specific information, e.g. rate of change in process variables, to help narrow down the diagnosis. Boolean reduction of this tree then yields the possible causes of the system fault, ranked according to their probability of occurrence, so that the most likely cause is shown first [61].

One traditional limitation of fault propagation models is the difficult of representing dynamic systems, i.e. systems in which the structure or behaviour can change in response to failures or as a result of normal system operation. Most fault propagation models are inherently static models and cannot readily represent the failure behaviour of a dynamic system, since the causes, effects, and propagation of a failure can be different in one state to the next.

This can be overcome in a number of ways, e.g. through the use of Dynamic Fault Trees or Pandora. It can also be overcome using a dynamic model that captures the behavioural and structural transformations that occur in complex dynamic systems [62]. The dynamic model is a hierarchy of abstract state machines developed around a structural model of the system, therefore representing both the behavioural and architectural aspects of the system. The lowest layers of the behavioural model indicate the deviation of subsystems or components from their normal behaviour and higher layers show how these deviations combine to produce more systematic failure transitions elsewhere in the system and ultimately cause a system failure [63]. Some of
these transitions in the state machines represent the top events of fault trees which capture the propagation of failure through the architecture of the corresponding subsystems. These fault trees can be automatically generated, e.g. using HiP-HOPS.

Figure 22 - Fault Trees as part of a dynamic behavioural model

 Unlike classical fault trees or CCDs, these dynamic fault propagation models are better equipped to record the gradual transformation of small scale failures into system malfunctions, taking into account both the dependencies and sequences of events involved. Such models can also be mechanically transformed into an executable specification that could be used with an automated real-time monitor [64].

Another approach to diagnosing dynamic problems using fault trees is to first convert them into decision trees known as Diagnostic Decision Trees (or DDTs) [65]. DDTs are binary trees that can be used for classification, in this case for faults or deviations, that pose a series of true/false questions to narrow down the diagnosis. As well as being constructed directly from static fault trees, DDTs can also be trained on historical system telemetry to predict faults and unknown faults, adding a dynamic capability to them. This also means that DDTs are suitable for use with real-time systems, as the decisions made are based on quantitative measurements and the binary nature of the trees means they are very efficient. Furthermore, DDTs can be used with Dynamic Fault Trees (DFTs) for an even greater capability for the analysis of dynamic systems [66].

3.3.2 Causal Process graphs

In addition to fault trees and CCDs, it is also possible to use ‘qualitative causal process graphs’. Like CCDs and fault trees, these attempt to model the failure behaviour of the system being monitored, but instead of describing the propagation of failure, they describe the interactions between system processes over the course of a failure or disturbance (though the interactions are not necessarily exclusive to disturbed behaviour). Such approaches include digraphs [67-77], logic flow graphs [70-79], and event graphs [72-82].

Digraphs (directed graphs) model the effect that qualitative changes in process parameters can have on other process parameters. Digraphs take the form of a graph of nodes - representing parameters - and connecting arrows, which represent relationships between those parameters. If
one parameter affects another, they are connected with an arrow (from the 'independent' parameter to the 'dependent' parameter) and a sign (i.e. +1, -1) which indicates the type of relationship. A positive sign indicates that if the independent parameter increases, then the dependent parameter will increase and vice versa, whereas a negative sign indicates an inverse relationship, i.e. if the independent parameter increases, then the dependent parameter decreases, and vice versa. Diagnostic systems can then make use of this information to trace the propagation of value deviations through the digraph, e.g. if parameter C is increasing and the digraph indicates that parameter B is inversely related to it, and observations show that B is decreasing, then whatever is causing B to decrease may in turn also be causing C to increase. Then the increase in C may in turn cause an increase in another parameter D etc.

![Figure 23 - An example Digraph](image)

Diagnosis relies on the ability of the monitor to locate one or more nodes ('primary deviations') that explain an observed deviation in a parameter's value. In practice, however, not all nodes are monitored and thus the diagnosis is likely to result in multiple possible primary deviations, any of which may be responsible for the observed symptom [52]. For example, the DIEX expert system [75] uses a backwards search on the digraph that terminates when the boundary of the digraph is reached, a normal node is met, or the same node is reached twice. The set of abnormal nodes is then stored and with each subsequent deviant node, the diagnosis is narrowed down further. The system can then report a list of potential faults associated with each primary deviation.

The general process of the digraph diagnosis approach is therefore as follows: firstly, all the component failures of the system are listed and the system is separated into sub-units; all control loops are then identified and classified (if present). Next, digraphs are generated for each sub-unit, and then these are joined into a system digraph by connecting common variables. This system-level digraph can then be used for determining the cause of faults: firstly, deviations are identified by comparing predicted behaviour against expected behaviour; then the diagnosis traces these deviations back through the digraph until no further tracing is possible. The results are the possible root causes of the problem [61].

Another version of the digraph approach is to use a fault tree that has been generated from the digraph as the diagnostic model [76]. When a deviation in a parameter is observed, it becomes the top event of a fault tree that is generated by searching through the digraph recursively until all primary deviations have been detected. The basic events of the fault tree represent the primary deviations, i.e. causes of the deviant parameter. The cut sets (i.e. reduced expressions) of the fault tree can then be used as part of an online monitoring system. However, this approach is only applicable to relatively simple systems without control loops, although more complex versions are possible that overcome this limitation, e.g. [77].

A further limitation of the digraph technique is the difficulty in developing digraphs for large processes. To overcome this limitation, it is possible to produce digraphs automatically from smaller digraphs of basic plant units [78]; the overall digraph is therefore synthesised from the
topology of the plant and a library of models represent common components, such as valves, pipes, and pumps etc [79-87].

Extensions to the traditional digraph notation have also been proposed. One example is the 'logic flowgraph' [70], which maintains the nodes and edges of digraphs but also introduces an extended set of symbols and rules to enhance flexibility and expressiveness. Logic flowgraphs are able to represent continuous and binary state variables, for example, while logic gates and condition edges can also be used to show how variables can be affected by combinations of conditions. Logic flowgraphs have also been extended to take into account dynamic elements in 'dynamic flowgraphs' [71-89] and were used in the development of a flight control system for the Titan II Space Launch Vehicle. Digraphs have also been extended to include some measure of quantitative information with the introduction of piece-wise linear transfer functions (QTF) and difference equations, which reduces the number of spurious results, and have been further extended to allow dynamic analysis too with the incorporation of the CUSUM method, which evaluates these quantitative measurements [83, 84].

The digraph is not the only possible qualitative causal process graph. MIDAS (Model-Integrated Diagnostic Analysis System) makes use of 'event graphs' to perform monitoring, validation, detection, and diagnosis [72]. Nodes in event graphs continue to represent the state of system parameters, but edges do not simply represent directional changes in values; instead, the indicate events and depict relationships between different states of the same variable ('intravariable links') or between different states of different variables ('intervariable links'). Event graphs can be built semi-automatically from digraphs. This approach can diagnose multiple faults provided that each fault influences separate measurements (i.e. the faults are independent and do not have similar effects) [75]. This is a significant advantage over traditional digraphs, which have been limited to hypotheses of single faults.

Because they focus on process variables and models describing how disturbances propagate between those variables, digraph-based approaches are often described as being process-oriented diagnosis techniques. This focus on processes can complicate the modelling and monitoring of complex control and operating procedures, however, and typically digraphs have been confined to relatively simple automated plant processes. Despite this, digraphs have been applied to larger systems in combination with other methods, e.g. statistical techniques [85], and as mentioned above it is in certain cases possible to diagnose multiple faults using digraph-style approaches [86].

### 3.3.3 Functional decomposition

The other paradigm of model-based diagnosis revolves around a model of the normal behaviour of the system rather than a model of the failure behaviour of the system. One type of model already mentioned is the Goal Tree Success Tree (GTST). A GTST describes the decomposition of a high-level safety objective into sub-goals and combinations of conditions that guarantee those goals will be achieved. The typical structure of a GTST is shown below:
Figure 24 - Goal Tree Success Tree (GTST)

GTSTs are logical trees containing only AND and OR gates. A monitor can therefore use simple propagation algorithms to update nodes of the tree that are not directly observable and to provide alarms for critical functions that fail. Furthermore, simple depth- or breadth-first searches can be carried out on the tree structure to locate the causes of functional failures. The flexibility of GTSTs has meant that they have been combined with other techniques, e.g. inference engines [87] or 'belief values' and probabilistic analysis [51], and GTSTs have successfully been used in a number of realistically complex applications in the nuclear [51], space [88], and petrochemical industries [89-97].

An alternative model is the 'Multilevel Flow Model' (MFM), which, like GTSTs, models the normal functional behaviour of the system [91]. MFMs describe the system goals (what it is meant to do) and the functions it uses to achieve those goals; to accomplish this, MFMs three types of entities: goals, functions, and components. The model is then presented as a set of goals which are accomplished by a number of functions which in turn are supported by a set of components. There are also a large number of possible 'flow functions', e.g. for mass, energy, and information, which further elaborate on the functional behaviour of the system.

MFMs can then be used with backward chaining techniques to test whether a hypothesis is true [92]. Once a goal has failed, the monitor hypothesises the failure of one or more of the functions that was to have achieved that goal, and then works backwards to determine which components supported those functions. MFMs are particularly useful in this regard for diagnosis with real-time constraints, because the hierarchy allows an imprecise diagnosis to be reached quickly and any further time can be used to refine the hypothesis to produce a more accurate diagnosis.

3.3.4 Behavioural models

Another model-based diagnosis approach is the 'diagnosis from first principles' approach, which uses a model of the system structure and its normal behaviour to make predictions about the state of the system operation. As opposed to GTSTs and digraphs, this form of diagnosis relies on the difference between the predicted parameters of the system model and the observed parameters of
the physical system. The logical system model describes the normal behaviour of the system and can be used to produce a set of possible hypotheses about system or component malfunctions to explain deviations between the observed and predicted behaviour [93-100].

This approach makes explicit use of three distinct stages in model-based diagnosis: hypothesis generation, hypothesis testing, and hypothesis discrimination [95]. Once a deviation from normal behaviour has been established, the first stage involves determining which components may be responsible for the deviation (hypothesis generation). These components are known as 'suspects' and in the second stage (hypothesis testing), the number of suspects is reduced by a process of elimination: each are hypothesised to have failed and the model is checked to see if this hypothesis is consistent with the observed behaviour of the system. If so, the hypothesis is held to be true, otherwise it is assumed to be false. Finally, if there are multiple remaining suspects, there must be some method of discriminating between them by making additional observations of the system, and this is the third stage.

Hypothesis generation can be simplified by using the structure and properties of system components to limit the number of suspects, e.g. by assuming the suspect is related or connected in some way to the deviation. The problem can be further simplified by exploiting measurements that indicate multiple discrepancies between observed and predicted behaviour, i.e. if there are two discrepancies detected, and there is a common suspect for them both, then it is more likely that this one common cause is the culprit rather than assuming two separate faults have occurred.

For hypothesis testing, there are two types of approach: 'constraint suspension', in which the behaviour of each component is modelled as a set of constraints, and 'assumption-based truth maintenance systems'. Constraint suspension involves defining all possible relationships between inputs and outputs of the system, allowing inferences to be drawn about one component if the state of another is known. This effectively transforms the model into a constraint network where values propagate in any direction; for example, the constraints of an addition function $A = B + C$ implies three constraints: $A = B + C$, $B = A – C$, and $C = A – B$. As long as sufficient measurements are available, this constraint network can be used to predict the state of non-observable components or parameters in the model. Its benefit to diagnosis stems from the fact that if the input measurements to the network are abnormal, then inconsistencies will arise as the expected results differ from the observed results; e.g. if $B = 2$ and $C = 2$ but $A = 5$, there is a discrepancy.

When testing a hypothesis, it is assumed that the suspect behaviour (whether normal or faulty) is unknown, and so its constraints are suspended (any input and output is possible). Other known values (determined from observation) are then fed into the constraint network and the resulting values propagate through it. If the network contains inconsistent values, e.g. one path suggests one value but a different path results in a different value, then the fault must lie somewhere else as the suspension of constraints did not remove the inconsistency. This is akin to a process of elimination – the faulty part is temporarily 'removed' but if it does not fix the problem, then it probably was not the cause. In which case, the hypothesis is assumed false and a different suspect is tested. If however the inconsistencies are resolved by suspending the constraints of the suspect, then it can be concluded fairly safely that the suspect was indeed the source of the discrepancy. The more observations fed into the network, the more accurate and precise it becomes.

Assumption-based truth maintenance systems utilise a different approach. One example is GDE (General Diagnostics Engine) [96-103] which provides a single mechanism for hypothesis generation and testing. The principle behind this approach is a predictive engine that propagates both values and their underlying 'assumptions'. For example, if all values are generated from a model of normal system behaviour, then the assumption underlying those values is that the system (or component) is functioning correctly.

In GDE, whenever discrepancies (or 'conflicts') are discovered, a set of suspects is generated by combining the assumptions underlying the conflicting predictions. As further conflicts occur, the set of suspects is reduced to leave only those suspects that can explain all detected conflicts. As a
result, this mechanism can deal with diagnoses of multiple faults as well as single faults. GDE has been successfully tested in the troubleshooting of digital circuits [98].

Although 'diagnosis from first principles' have been successfully applied to simple systems, including GDE and KATE [99], which was developed for NASA, the approach is less suitable for the diagnosis of dynamic or complex systems. For example, constraint propagators rely on snapshot measurements and do not generally take into account the history or sequences of measurements. Other problems involve the difficulties in representing and reasoning about complex behaviour, particularly with regards to the adequacy and efficiency of the predictive engines involved [92, 65, 66].

3.3.5 Qualitative simulation

An alternative approach for the fault diagnosis of dynamic systems is qualitative simulation. In this approach, the system model – typically represented as a set of quantitative differential equations – is transformed into a set of 'qualitative constraints', e.g. representing equalities. One example of this approach is the QSIM tool [101], which simulates a system that is represented as a set of qualitative constraints. QSIM can predict qualitative values (i.e. increasing, stable, decreasing) and quantitative ranges for each parameter. QSIM has been used in MIMIC [102] for the diagnosis of faults in dynamic devices. When an anomaly is detection in the system, a set of hypotheses are generated and a QSIM model is initialised for each one. If the predictions of the QSIM model do not match the observed behaviour of the system, the hypothesis is discarded.

One drawback of QSIM is that it requires one model for each of the potentially large number of possible faults. This also means that unanticipated faults are not modelled and would go undetected [103]. However, there have been attempts to overcome this using algorithms that integrate qualitative simulation with diagnosis from first principles [104]. In these cases, the normal behaviour is specified using qualitative constraints and thus QSIM can be used as the predictive engine.

Rodelica is another equation-based approach, this time a declarative object-oriented language derived from Modelica and adapted to model-based diagnosis [105]. In addition to qualitative representations, it allows a greater degree of numerical representation than QSIM by offering interval ranges and sets of discrete values. Relations between different values are represented using a greater range constraints than simple equalities, including inequalities, conditional constraints, and Boolean relations; furthermore, both the normal and failure behaviour can be defined for each class in the language, as each class in the language can define a set of failure modes. Once modelled in Rodelica, the system model can then be used for diagnosis by the RODON reasoning engine [106], which functions along the lines of diagnosis from first principles – conflicts between predicted and observed behaviour are detected, and hypotheses are generated and tested to explain these conflicts.

The Rodelica / RODON approach has the benefit that it models both normal and malfunctioning system behaviour, enabling it to detect more subtle faults including the occurrence of multiple faults. As a consequence, however, it is also a very expensive technique. There are a number of ways to overcome this; one method is to encourage additional input from the user, and the Rodelica / RODON combination offers an 'interactive model-based diagnosis' (IMDB) mode in which the user provides further measurements to progressively refine the diagnosis. Another method, where resources for diagnosis are more limited, is to generate decision trees as a more compact form of diagnostic knowledge. A decision tree can be used to determine which system state explains a symptom; the symptom is the root of the tree and leaf nodes describe the faulty states that explain it. In between are intermediate decisions nodes (e.g. involving a test or measurement) to help determine which leaf node is the correct one. RODON can use the information provided by the Rodelica model to generate decisions trees automatically; however,
this involves a systematic simulation of all relevant operational states to produce a 'state database' (SDB), and from this the decision trees are formed. A side effect of this process is that the SDB can also be used for risk analyses such as FMEA.

### 3.4 Data-driven approaches

In addition to rule-based and model-based fault diagnosis, there is a third paradigm: data-driven approaches. In these approaches, a predictive model of the normal system behaviour is constructed from empirical data collected from the operation of the system. The model therefore acts as a function that relates the collected data to the different parameters of the system, e.g. the relation between output and input parameters. The advantage is that the predictive model can be generated without in-depth knowledge of the system, only from a historical record of observations, but the difficulty is in finding a function that generates accurate predictions from past data.

There are three general types of data-driven approaches: statistical process monitoring, monitoring of qualitative trends, and neural networks. The first type, statistical process monitoring, can be further subdivided into 'univariate' and 'multivariate' methods. In univariate methods, the fitting function typically relates only one dependent variable (e.g. the output parameter) to multiple independent variables (e.g. inputs and controls). This is suitable for smaller, simpler systems, in which the fitting function can correspond well to the empirical data, but in more complex systems, higher order non-linear functions may be required and it is difficult to model these with univariate modelling. Furthermore, the accuracy of univariate methods depend strongly on the quality of the measurements for the independent variables.

Multivariate approaches instead make use of statistical methods and sets of historical data to create more accurate models that can better predict both dependent and independent variables. Examples include principal component analysis (PCA) [107] and partial least squares [108]. The discrepancy between predicted and observed values can then be used in real time to help locate possible disturbances in the system processes.

One prominent method of multivariate process monitoring is the use of classification and regression trees (CART), which enables the analysis of large data sets via binary partitioning; it is similar to the binary DDTs described earlier. CART operates by recursively partitioning the dataset into smaller and smaller subsets, making it possible to base conclusions on the subset in which data appears. It can also be combined with Fisher Discriminant Analysis (FDA), which reduces the dimensionality of the process measurements by simplifying the discrimination between different classes of measurements, improving the accuracy and efficiency of the CART method [109].

The second type is of data-driven fault detection is qualitative trend analysis [110]. This approach consists of two main steps: the identification of trends in the measurements and the interpretation of those trends in fault scenarios. Parameter trends can be constructed from different sets of primitives, each describing the qualitative state of a parameter, e.g. 'stable', 'increasing with an increasing rate', etc [111]. A sequence of these primitives makes up a trend and can be used to monitor the parameters of the system for deviations from expected trends.

The third type is based on the use of neural networks, which are trained on the normal operation of the system; diagnosis is then based on the discrepancy between the predictions of the neural networks and the observations of the system in operation. Neural networks have a number of advantages over qualitative trend analysis, including a better tolerance for noise and the ability to learn from examples. Both neural networks and particularly qualitative trend analysis are capable of dealing with large amounts of process data [112].
4 Safety design flow

With the trend of increasing complexity, software content and mechatronic implementation, there are increasing risks from systematic failures and E/E random hardware failures, often rooted in management and engineering processes. The new International Standard ISO 26262 includes guidance to avoid these risks by providing feasible requirements and processes. This International Standard is the adaptation of IEC 61508 to comply with needs specific to the application sector of E/E systems within road vehicles. This adaptation applies to all activities during the safety lifecycle of safety-related systems comprised of electrical, electronic, and software elements that provide safety-related functions.

The ISO 26262:
- Provides an automotive safety lifecycle (management, development, production, operation, service, decommissioning) and supports tailoring the necessary activities during these lifecycle phases;
- Provides an automotive specific risk-based approach for determining risk classes (Automotive Safety Integrity Levels, ASILs);
- Uses ASILs for specifying the item's necessary safety requirements for achieving an acceptable residual risk; and
- Provides requirements for validation and confirmation measures to ensure a sufficient and acceptable level of safety being achieved.

The main safety lifecycle phases are shown in the figure below:

![Figure 25 - Main phases of the safety lifecycle](image)

The safety activities encompassed in the concept phase and system design phase will be detailed in the following paragraphs, by describing the safety design flow, which includes the tasks necessary to perform a safety analysis compliant with ISO CD 26262.
System functional analysis

System vehicle interaction analysis

System boundary definition

Normative requirements

Hazard identification

Risk analysis

Safety Goals & Safe States definition

Risk assessment approval

ASIL=QM

NO

SI

STOP ANALYSIS

Functional safety requirements definition

ASIL(s) Allocation to functions/components

Technical safety requirements definition

System architecture review
In the following a detailed description of each step of the safety design flow will be given.

4.1 System functional analysis

The first step of safety lifecycle consists to identify and describe the Item under safety analysis, and to develop an adequate understanding of it. This is an essential step, since the subsequent phases of safety design flow are based on the item definition and the safety concept is derived from it.

To have a satisfactory understanding of the item is necessary to know information about its functionality, interfaces, environmental conditions etc. By analysing all the available technical documentation concerning the item, it is possible to take out the system functions list and their description, as well as the item’s input/output(s) and the functional interactions among the main components/subsystems involved on board the vehicle.

Furthermore, by means of the UML use case formalism adapted to the safety field, it is possible to collect in an exhaustive way the interactions between the external environment and the item itself. The function is defined from the Actor point of view. Actors are external entities (people or other systems) which interact with the system to achieve a desired goal.

- “Use case” is a scenarios set; all this scenarios have in common the final aim of user.
- “Scenario” is a flow of actions and interactions between system and actors.

Each (safety oriented) use case is described by specifying:

- **System name**: name of the System/project Under Discussion (SUD)
- **Use Case name**: name of the use case
- **Short description**: short description of the main goals of the use case
- **Target Function(s)**: the function description in terms of output(s) behaviour
- **Primary actor**: main user of the SUD
- **Secondary actor(s)**: takes advantages from the SUD information but it isn’t active into the specific use case
- **Pre-condition(s)**: All the condition to be verified at the beginning of the use case
- **Application scenario**: application scenario: driving situation (def. WD26262: “scenario that may occur while a vehicle is in use-moving or stationary”) and environmental condition (def. WD26262: “Physical or other constraints under which an item is used”)
- **Operational scenario**: Sequence of actions and interactions among the system and one or more actors

17 The complete and correct definition of function includes the system output(s) identification. For each identified output the corresponding target function shall be defined.

Target function examples:

a) "Request to VDC [vehicle dynamic control system] for a braking torque to the rear inner wheel." \(\rightarrow\) if the output of the function under analysis is linked to VDC for the physical braking actuation, it should be wrong to define the target function in terms of "braking actuation" since the final physical actuator is managed by another system [VDC] outside the function’s boundary.

b) "engine cranking actuation" \(\rightarrow\) if the output of the function under analysis [i.e. Start&Stop] is in charge to directly manage the relay for the engine crank.

The correct target function(s) definition is fundamental for a correct statement of risk analysis, and in particular for the malfunction(s) definition.
- **Fail condition(s):** malfunctions - all different possible termination of the ability of the functionality to perform a function as required
- **Misuse(s):** incorrect, improper, or careless use of the SUD
- **Risk’s source:** Origin of the Fail condition/misuse
- **Function Criticality(s):** Criticality of the function, related to the use case, due to external factor(s)
- **Post-condition:** describe the condition in which the SUD will arrive if the system flow is correct
- **Status:** description of the use case status (to be approved, approved, in modification,...)
- **Open issues:** any issues which require discussion affecting this use case
- **Comments:** any comments on the contents of the use case.

The above data are collected in the Use case Table (see Table 1).

Moreover, for the completion of the knowledge about the item functionality, it is essential to understand the system’s internal logic carried on to fulfil the functional specifications. In particular, by defining a data flow diagram of the item, it is possible to describe and analyse the flow of data internally, through the processes of the item, and externally, through the interfaces to external system.

To highlight the interactions of the system (sub-system, function, etc) under safety analysis with other sub-systems or modules, it is necessary to produce a block diagram that is an architecture schematisation from which boundary elements, interfaces and communication lines can be easily determined.

In the block diagram representation the interfaces are defined in terms of functional devices (to environment) and ports (to EE internal entities).
# Table 1: Use case table

<table>
<thead>
<tr>
<th>#</th>
<th>System Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Use case name</td>
</tr>
<tr>
<td>2</td>
<td>Case ID</td>
</tr>
<tr>
<td>3</td>
<td>Short description</td>
</tr>
<tr>
<td>4</td>
<td>Target Function</td>
</tr>
<tr>
<td>5</td>
<td>Primary Actor</td>
</tr>
<tr>
<td>6</td>
<td>Secondary Actor(s)</td>
</tr>
<tr>
<td>7</td>
<td>Pre-condition(s)</td>
</tr>
<tr>
<td>8</td>
<td>Application Scenario (Driving situation + Environmental conditions)</td>
</tr>
<tr>
<td>9</td>
<td>Main flow events (or operational scenario)</td>
</tr>
<tr>
<td>10</td>
<td>Fail condition(s)</td>
</tr>
<tr>
<td>11</td>
<td>Misuse(s)</td>
</tr>
<tr>
<td>12</td>
<td>Risk’s source</td>
</tr>
<tr>
<td>13</td>
<td>Function Criticality(s)</td>
</tr>
<tr>
<td>14</td>
<td>Post-condition(s)</td>
</tr>
<tr>
<td>15</td>
<td>Status</td>
</tr>
<tr>
<td>16</td>
<td>Open issues</td>
</tr>
<tr>
<td>17</td>
<td>Comments</td>
</tr>
</tbody>
</table>
4.2 System-Vehicle interaction analysis

The next step of the safety workflow consists of analysing the item/vehicle interaction and how the new functionality impacts on the vehicle behaviour, i.e. in terms of lateral and/or longitudinal stability. To perform these kinds of analysis the behavioural model of the functions should be defined. Through the behavioural model definition it is possible to analyse the functional content by providing an algorithmic description of behaviour. Examples of behavioural model are the state machine models or Simulink models that, together with data processing models (e.g. data flow diagram), are required to describe the system’s behaviour from different perspectives.

An impact analysis should also be done to determine which parts of an already existing vehicle architecture will be affected by the new item implementation, in terms of changes and consequences. For this purpose all the components/sub-systems, related and/or interfaced to the item under safety analysis, should be listed and described, as well as the status of their current development (e.g. Standard Production or New development).

The proven in use argumentation is another useful information to collect in this phase. A product (component, sub-system, etc) identical to an existing one or at least which has a very high degree of commonality with an existing product is named proven in use. Proven in use argument consists of an appropriate documentation on the proven in use product, its change management, configuration control and field data regarding its safety characteristics (ISO CD26262, part 8).

4.3 System boundary definition

As required to the new standard (ISO CD 26262, Part 3), it is necessary to provide a solid understanding of the way the item interacts within its surroundings. Therefore, the boundary of the item and the item’s interfaces with other elements must be determined. The functions implemented in the item could be classified in three kinds of classes:

- **Standard Production (SP)**: these functions are invariant regarding the current vehicles and they will be the basis on which the item system will be implemented and, in terms of reliability and safety, they shall be compliant with the current Standard Production.

- **Safety Critical System (SCS)**: these SCS functions are potentially “Safety Critical”; in other terms these functions shall be analysed in deep in order to evaluate the “Safety Integrity Level” and in consequence of it to define the robustness.

- **Reliable System (RS)**: these functions are considered reliable, and they haven’t impact with the safety

Then, taking into account the potential safety criticality of the functions, the boundary of the Subsystem Under Safety Analysis (SUSA) shall been selected.

An example of SUSA definition is shown below:
In order to perform a correct hazard analysis, a proper set of realistic operative scenarios must be defined. A complete description of a scenario includes all variables and/or states that characterise the functions or affect them.

A scenario includes:

- **Operative conditions**: selection of the factors related to vehicle usage scenarios, risen from driver choices to be considered for situation analysis and hazard identification. For instance, in vehicle dynamic control field, the following are typical operative conditions families:

  - **Dynamic driving state**
    - Speed
    - Longitudinal acceleration
    - Lateral acceleration
    - ...

  - **Manoeuvres**
    - Curve
    - Overtaking
    - Braking
    - Lane change
    - ...

  - **Driver condition**
    - Attention level
    - Hands on steering
    - ...

### Table 2: SUSA Perimeter

<table>
<thead>
<tr>
<th>ID</th>
<th>Subsystem</th>
<th>Description</th>
<th>SUSA Perimeter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PBAU</td>
<td>Parking brake</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>VDB</td>
<td>Vehicle Data Bus: High speed CAN (250 Kbit/s)</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>Gearbox</td>
<td>Clutch pedal status</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>Body</td>
<td>Door status</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>ABS/ASR/ESP</td>
<td>Wheel speed</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Engine</td>
<td>Engine status</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>Power Device Supply</td>
<td>Battery</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>HMI</td>
<td>Lamp Fault + Lamp Status + Buzzer</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2010 The ATESTT2 Consortium
- **Environmental conditions**: selection of the factors related to environmental variables to be considered for situation analysis and hazard identification. Typical environmental conditions are, for example, the following:
  
  o **Location**
    ✓ Urban road
    ✓ Suburban road
    ✓ Motorway or highway (with lateral barriers)
    ✓ …
  
  o **Street condition**
    ✓ Adherence
    ✓ Road surface type (smooth, …)
    ✓ Slope
    ✓ …
  
  o **Traffic situation**
    ✓ Presence of other vehicle (lateral, on coming, following, …)
    ✓ Presence of cycles, pedestrians, etc.

In coherence with the item functionality under analysis, the depth of detail has to be sufficient to guarantee a realistic evaluation of the effects, in terms of vehicle behaviour and driver’s control capability, forced by a malfunction. Usually, physical quantities, such as velocity or acceleration or radius of curvature, are split into several ranges of interest.

As a concluding remark, the scenarios definition, as well as the subsequent hazards identification activity, requires a clear definition of the item, in terms of its functionality, its target function and its boundary. Up to now, these phases of analysis are based on the item’s behaviour only, even in absence of a deeper knowledge about its detailed hardware/software design.

### 4.5 Hazard identification

To identify the hazards it is essential to define the malfunctions, the misuse(s) and eventual maintenance condition(s) related to the item.

#### 4.5.1 Malfunctions definition

All possible functional anomalies derivable from each foreseeable source, either internal (system faults) or external (e.g. foreseeable misuse) shall be defined.

In general, for each target function is always possible to define, in exhaustive manner, the malfunctions in terms of anomalies of function activation:

- Unwanted activation
- Missed activation
- Degraded activation (vs. value or timing)
- Incoherent activation (in presence of more than one function’s output)
- Unwanted deactivation
- Missed deactivation
- Degraded deactivation (vs. value or timing)
- Incoherent deactivation (in presence of more than one function’s output)
Note: as already previously mentioned, the malfunctions are to be declared as anomalies of item’s outputs (not necessarily coincident with the vehicle outputs, unless the physical actuators are “proprietary” of the item under analysis).

### 4.5.2 Misuses and maintenance condition definition

All foreseeable misuses (or error in use) related to the item shall be defined as well as foreseeable risky operational during maintenance in garage or at fuel station (maintenance conditions). The anomalies caused by voluntary incorrect use of the item are generally excluded.

### 4.5.3 Hazard definition

Hazard is a potential source of harm, due to an item malfunction in concomitance with a particular scenario’s condition. Therefore a malfunction can be cause of several hazards with different consequences, really as a function of the involved scenario.

![Diagram of Malfunction-Scenario-Hazard relation](image)

**Examples:**

- **Malfunction:** Unwanted automatic rear wheels braking actuation request
  - **Scenario [1]:** suburban road, medium-high speed \([V>70\text{kph}]\), high adherence, normal traffic on both directions, curve \([\text{medium-low radius}],\) medium-high lateral acceleration \([a_L>0.3\text{g}]\)
  - **Hazard [1]:** Unexpected braking action in curve \(\rightarrow\) a yawing moment is triggered \(\rightarrow\) vehicle instability due to the loss of lateral dynamics control \(\rightarrow\) lane changing or out of road
  - **Scenario [2]:** urban road, approaching a crossroad, medium-low speed \([v<50\text{kph}]\), normal traffic \([\text{vehicles, bicycles, pedestrians}]\)
  - **Hazard [2]:** Vehicle suddenly halted in middle of a crossroad \(\rightarrow\) possible accident with other vehicles, tram.
  - **Scenario [3]:** suburban road, low speed, approaching a railway crossing
  - **Hazard [3]:** Vehicle kept halted on a railway line in concomitance with the arrival of a vehicle that cannot stop \([\text{train}]\)
4.6 Risk analysis

The hazards, related to each malfunction/misuse evaluated in each relevant scenario, shall be classified in terms of Automotive Safety Integrity Level (ASIL) in order to estimate the risk level of the item. In the following paragraphs, a description of the steps needed for a risk assessment is reported.

4.6.1 Scenarios tree definition

The aim is to collect, for each target function and for each malfunction/misuse, all the potential risky scenarios by properly combining the operative and environmental conditions selected as relevant for the function under safety analysis.

At the end of this activity a scenarios tree is built up, where each “branch” of the tree is describing a dedicated scenario to be investigated since its potential safety relevance.

Take into account that a scenarios tree referred to a complex item could usually become “as big as an oak” since the amount of combinations easily leads to several hundreds of “branches”.

4.6.2 Controllability level estimate

The final aim of this section is the evaluation of possibilities to avoid harm. The requested “controllability” ranking is in effect an estimation of the probability that the driver [or, in general, any other involved actor] is able to gain control of the arising hazardous event in order to avoid harm.

The only reasonable approach is:

- To define a list of controllability “expected tasks” for avoiding or limiting the hazardous effect.
  Obviously these tasks are depending on the specific hazard and are referring to the normal operability of a average experienced driver [i.e.: countersteering, force increasing on braking pedal, switch emergency lamps on, crank again, etc.]

- To identify a specific criterion to be applied for the controllability classification.
  The criterion has to be selected by taking into account the physical phenomenon, triggered by the malfunction, whose deviation from the standard behaviour is the cause of hazard and potential harm. In general, the controllability level is always a function of the driver’s reaction time.
  For instance, if the hazard is caused by a poor controllability on vehicle’s dynamics then a safe displacement of normal operability around the vehicle could be established, let’s say about 20cm. On this hypothesis, any abnormal deviation [lateral or longitudinal] from the vehicle trajectory is considered as not safety relevant: the statistical driver’s response will address to a correct controllability ranking. Typically, if almost all drivers [99% of general driver population] are able to gain control [deviation from vehicle’s trajectory always less then 20cm] of the hazardous event, then the controllability is maximum. Otherwise other rankings are allowable, down to the minimum of controllability when a very small percentage of people are able to “rescue” the hazardous situation, of course apart from professional drivers.

- To estimate the controllability level with regard to the capability to carry out the expected tasks: classification addressed by the identified specific criterion.
Example:

**Hazard:** Missing Hill Holder actuation in high slope condition → vehicle moving as soon as the brake pedal is released.

**Expected task(s):** Press again the brake pedal down

**Specific criterion:** Driver’s reaction time → measurement of displacement as a function of the slope gradient.

For each malfunction/misuse and for each selected scenario the vehicle controllability tasks shall be defined, that are the expected timely reactions of the involved persons to avoidance specified harm or damage. Then, on the bases of controllability task considerations, for each malfunction/misuse the controllability level can be estimated.

ISO CD26262 defines the following classes of Controllability:

- C0: Controllable in general
- C1: Simply controllable
- C2: Normally controllable
- C3: Difficult to control or uncontrollable

The controllability evaluation can be carried on, in a preliminary phase, through company know-how and/or simulations. These preliminary estimates shall be confirmed by performing a specific testing, typically on a proving ground, using the fault injection technique.

### 4.6.3 Severity estimate

The severity level defines the degree of harm, in terms of physical damage to each endangered individual (including the driver or the passengers and all other traffic participants, such as cyclist or pedestrians, involved in the scenario). For this reason it is important that the scenarios definition always include the traffic situation description (into environmental condition).

To estimate the severity level for each malfunction, the accidental scenarios, if controllability tasks will fail (in other terms without any driver control), should be described. The scope is to provide sufficient information to foresee possible malfunction consequences in a specific traffic scenario, in order to classify the severity level.

The criteria for ranking an accident are usually coming from statistical data [data base from insurance companies, health or police departments, etc], or, in general, the expected seriousness of an accident is a rough estimate based on company know-how due to previous accident analysis. At first instance, severity categorisations normally suggested are based on different injury scales, such as AIS, ISS, NISS.

Up to now, examples of severity categorisation are in progress from ISO for addressing purposes only: they are mainly a function of class of accident [i.e. side or rear or front collision between cars, rollover, out of road, pedestrian/bicycles accident,...] and relative velocity between the actors involved.
The classification shall be done compliant with ISO CD26262 severity definition. ISO CD26262 defines four level of Severity:

- S0: No injuries
- S1: Light and moderate injuries
- S2: Severe and life-threatening injuries (survival probable)
- S3: Life-threatening injuries (survival uncertain), fatal injuries

### 4.6.4 Scenarios tree reduction

As requested by the ISO° 26262, the scenarios tree must be reduced, by aggregating mutually exclusive operational situations that have the same severity and controllability class, and by removing all irrelevant scenarios (Controllability class C0 or severity class S0).

In fact, detailing the operating situations for one hazard with regard to vehicle state, road and environmental conditions can lead to a very granular classification so to allow an easier rating of controllability and severity. Because of this approach the resulting situations may lead to an improper reduction in the classification of the exposure time, and finally a lower level of risk. So, to avoid such improper results, the aggregation of scenarios is required.

### 4.6.5 Exposure time estimate

Based on company experience and statistical data, for each scenario an estimate of how frequently and for how long individuals find themselves in a situation where the hazard may exist (E = probability of exposure) should be provided.

The ISO CD26262 define the following class of Exposure:

- E1: Very low probability
- E2: Low probability
- E3: Medium probability
- E4: High probability

### 4.6.6 ASIL determination

For each potential hazard situation the Automotive Safety Integrity Level should be determined. The ASIL level specifies the item’s necessary safety requirements for achieving an acceptable residual risk. A risk (R) can basically be described as a function $F$ of the frequency ($f$) of occurrence of a hazardous event and the potential severity of the resulting harm or damage ($S$ = severity):

$$R = F(f, S)$$

The frequency of occurrence $f$ is in turn influenced by controllability and exposure ($f = E \times C$).
The ASIL level shall be determined by combining, for each Hazardous event, Severity, Controllability and Exposure Time, in accordance with the following table (ISO CD26262, Part 3):

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>E1</td>
<td>QM</td>
</tr>
<tr>
<td></td>
<td>E2</td>
<td>QM</td>
</tr>
<tr>
<td></td>
<td>E3</td>
<td>QM</td>
</tr>
<tr>
<td></td>
<td>E4</td>
<td>QM</td>
</tr>
<tr>
<td>S2</td>
<td>E1</td>
<td>QM</td>
</tr>
<tr>
<td></td>
<td>E2</td>
<td>QM</td>
</tr>
<tr>
<td></td>
<td>E3</td>
<td>QM</td>
</tr>
<tr>
<td></td>
<td>E4</td>
<td>A</td>
</tr>
<tr>
<td>S3</td>
<td>E1</td>
<td>QM</td>
</tr>
<tr>
<td></td>
<td>E2</td>
<td>QM</td>
</tr>
<tr>
<td></td>
<td>E3</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>E4</td>
<td>B</td>
</tr>
</tbody>
</table>

Table 3: ASIL determination (from ISO CD26262, part 3)

The ISO CD26262 defines four ASIL level: ASIL A, B, C and D, with D representing the highest and A the lowest class. In addition to these four levels, the new standard defines the level QM (Quality management). Only the functions that have been considered not safety relevant can belong to QM class.

4.7 Safety Goals and Safe States definition

If the item under safety analysis has only hazards classified as QM, then the safety analysis can be stopped. If, instead, the item cannot be classified as Quality Management, then the Safety Goals for each hazardous event should be defined.

Safety goals are the Top-level safety requirements. It leads to item characteristics needed to avert the hazard or to reduce risk associated with the hazard to an acceptable level. Therefore an ASIL shall also be assigned to each safety goal, to provide evidence that the goal has not to be violated (by hardware faults) with a probability higher than the value assigned to the associated ASIL level.
Similar safety goals can be combined into one safety goal and, if different ASILs are assigned to similar safety goals, the highest ASIL shall be assigned to the combined safety goal (see Safety Goals definition example). Moreover, for every safety goal, a safe state should be defined, in order to declare a system state to be maintained or to be reached when the failure is detected and so to allow a failure mitigation action without any violation of the associated safety goal.

To better clarify the above concepts, an example of safety goals assignment is given below.

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>Malfunction</th>
<th>Safety Goal</th>
<th>Associated Safe State</th>
<th>Associated ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario1</td>
<td>M1</td>
<td>SG1</td>
<td>SS1</td>
<td>A</td>
</tr>
<tr>
<td>Scenario2</td>
<td>M1</td>
<td>SG1</td>
<td>SS2</td>
<td>C</td>
</tr>
<tr>
<td>Scenario3</td>
<td>M1</td>
<td>SG1</td>
<td>SS2</td>
<td>A</td>
</tr>
<tr>
<td>Scenario1</td>
<td>M2</td>
<td>SG3</td>
<td>SS3</td>
<td>D</td>
</tr>
<tr>
<td>Scenario1</td>
<td>M3</td>
<td>SG4</td>
<td>SS4</td>
<td>A</td>
</tr>
<tr>
<td>Scenario2</td>
<td>M3</td>
<td>SG3</td>
<td>SS3</td>
<td>C</td>
</tr>
</tbody>
</table>

Therefore, a safety goal should be associated with each malfunction and an ASIL level and a safe state (if applicable) should be associated to each safety goal. In other words, ASILs and safe states should be safety goal attributes.
4.8 Risk assessment approval

The estimated controllability values assigned to the several situations should be validated. Therefore, specific testing on road shall be carried out for the final value assessment. The idea consists of performing a “Test Drive” by using a significant vehicle to inject the faults that determine the failures, and monitoring the drivers’ opinions and reactions. To formalise in a rational way the drivers’ opinions it is possible to use the “Cooper-Harper method” customised for the automotive field.

4.8.1 Cooper-Harper Scale

The Cooper-Harper scale was suggested in 1969 to evaluate the aircraft response related to a particular task or a particular request, by basing the evaluation on parameters like controllability and performances.

The Cooper-Harper Scale


Figure 28 – Cooper-Harper scale
By means of a decision making process through the proposed flow, the pilot was forced to the final evaluation.

The Cooper-Harper scale is split into ten levels, where the first one indicates an excellent evaluation and the last one indicates an uncontrollable aircraft. The same method can be adapted for the automotive field, to formalise the test drivers’ opinions in a rational way. The drivers should quantify our evaluations by following a decision path.

The questions put to the test drivers should be specific for the case under analysis. After each decision, the driver opinion is more detailed, and with each answer the driver implicitly assigns a vehicle controllability level. Since ten evaluation levels could be too much for collecting the controllability data, it could be useful to reduce the scale levels by ten to five. These five levels collect all ten controllability level defined in the Cooper-Harper scale.

The following picture shows an example of controllability assessment by using the Cooper-Harper scale customised on steering.

**Controllability valuation with Harper Cooper method customised on steering**

<table>
<thead>
<tr>
<th>Controllability Classification</th>
<th>Description of the controllability classes</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C4</td>
<td>The driver cannot drive the car</td>
<td>10</td>
</tr>
<tr>
<td>C3</td>
<td>The situation is very critical you can counter some with great effort and for a short time, the vehicle is out of control</td>
<td>9</td>
</tr>
<tr>
<td>C2</td>
<td>The driver managed to counter the trouble with a great effort, the vehicle is out of control</td>
<td>8</td>
</tr>
<tr>
<td>C1</td>
<td>The driver managed to counter the disturbance, control is compensate</td>
<td>7</td>
</tr>
<tr>
<td>C0</td>
<td>The reaction of the vehicle is perceived and the driver can barely control</td>
<td>6</td>
</tr>
<tr>
<td>C5</td>
<td>The reaction of the vehicle is perceived and the driver can control effort</td>
<td>5</td>
</tr>
<tr>
<td>C6</td>
<td>The reaction of the vehicle is perceived and the driver can control with minimal effort</td>
<td>4</td>
</tr>
<tr>
<td>C7</td>
<td>The disturbance is perceived interesting guidance</td>
<td>3</td>
</tr>
<tr>
<td>C8</td>
<td>The disturbance is perceived without attracting the guidance</td>
<td>2</td>
</tr>
<tr>
<td>C9</td>
<td>The disturbance is just perceived</td>
<td>1</td>
</tr>
<tr>
<td>C0</td>
<td>The information is not perceived</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 29 - Cooper-Harper example**

### 4.9 Functional safety requirements definition

For each safety goal and safe state (if applicable), results of the risk assessment, at least one safety requirement should be specified. The defined safety goals are the “Top-level functional safety requirements”.

© 2010 The ATESTSST2 Consortium
As specified in the ISO 26262 (Part 3), ASIL, operating modes, fault tolerant time spans (if applicable), safe states (if applicable), emergency operation times (if applicable) and functional redundancies (if applicable) shall be attributes of the functional safety requirements. To develop a complete set of effective functional safety requirements, the identification and analysis of all common causes of failure and single points of failure it’s necessary. This kind of analysis can generally be performed by means of System FMEA & FTA techniques.

Starting from the top level functional safety requirements, it is necessary to understand which parts of the safety channel are critical, by performing, for instance, a system FMEA. To satisfy the safety goals for each identified failure mode the diagnosis and recovery actions should be specified.

4.9.1 System FMEA vs Risk assessment

The Risk Assessment can be linked to the System FMEA. In fact, the FMEA will provide information to evaluate hazards (identified in the risk assessment), identify safety critical areas, and provide inputs to safety design criteria and procedures with provisions and alternatives to eliminate or control all unacceptable and undesirable hazards, based on their combination of severity and probability of occurrence. The link is more clear if we note that the hazards evaluated in the risk assessment are nothing but the “Vehicle potential effect of failure” defined in the System FMEA table (see following figure).

![System FMEA process](image)

**Figure 30 – System FMEA process**

4.9.1.1 Types of FMEA

The Failure Mode and Effect Analysis (FMEA) is a “logical, structured analysis of a system, subsystem, device, or process” (Schubert, 1992). It is one of the most commonly used reliability and system safety analysis techniques.

The purpose of a FMEA is to define the ways in which a failure can occur as well as the causes and resulting effects of a failure on a defined system.

A FMEA can be described as a systematic group of activities intended to:

- Recognise and evaluate the potential failure of a product/process and the effects of that failure
- Identify actions that could eliminate or reduce the chance of the potential failure occurring,
- Assist in selecting design alternatives with high reliability and high safety potential during the early design phases
- List potential failures and identify the severity of their effects
- Develop early criteria for test planning and requirements for test equipment
• Provide historical documentation for future reference to aid in analysis of field failures and consideration of design changes
• Provide a basis for maintenance planning
• Provide a basis for quantitative reliability and availability analyses.

Process FMEA:
Used to analyse transactional processes. Focus is on failure to produce intended requirement, a defect. Failure modes may stem from causes identified.

Design FMEA:
Used to analyse components designs. Focus is on potential failure modes associated with the functionality of a component caused by design.

The failure analysis of Design FMEA is carried out as following [ref. VDA – Quality Assurance before series production – System FMEA, Vol. 4, Part 2]:

• **Failure of the component**: the component is regarded as system element. The physical component failures under service conditions of production are considered as potential failure.
• **Failure Causes**: the failure cause are searched for every potential failure of a component by investigating its design data.
• **Failure effects**: the failure effects, such as the effects of wear, lead to malfunction of the examined product.

System FMEA (Design potential FMEA):
It focuses on potential failure modes associated with the functionality of a system caused by design.

The failure analysis of the Design potential FMEA is carried out as following [ref. VDA – Quality Assurance before series production – System FMEA, Vol. 4, Part 2]:

• **Failure of the system**: a system is examined, which consist of multiple components. This system does possibly not meet one of its functions under service conditions. The malfunctions are considered as possible failures by the Design potential FMEA.
• **Failure causes of the system**: the possible failure causes are searched in the malfunctions of the system elements.
• **Failure effects within the system**: the possible failure effects lead to the malfunctions of the superior system elements.

Design potential FMEA (System FMEA) regards possible malfunctions of product systems as possible failures. If necessary, the failure analysis goes down to the design failures of individual components step by step (Component FMEA). Afterwards, it is possible to perform a FMEA for each level of analysis. The FMEAs carried out on different levels link each other in complementary way (see fig. …). In particular:

• The potential failure mode of the upper level (System FMEA) is carried over as the potential failure effect for the FMEA of the next lower level (Component FMEA).
• A potential failure cause in an upper level can be carried over as a potential failure mode in the next lower level.
Figure 31 – Component FMEA process

4.10 Technical safety requirements definition

Once the functional safety concept is specified, the item can be developed at the system prospective, starting from the technical safety requirements specification.

The technical safety concept (that consists of the technical safety requirements) describes how to implement the safety measures described in the functional safety concept. The objective is to detail the item-level functional safety requirements into system-level technical requirements, down to the separation between hardware and software (ISO CD26262, part 4). In other terms, the technical safety requirements use the functional safety requirements to describe how the technical architecture and the parts of the system will fulfil the system requirements.

The technical safety requirements have to be defined during architecture and system design and measures for fault avoidance (systematic faults and hardware random faults) and mitigation have to be described during hardware and software components design.

The process of technical safety requirements definition is an iterative process that passes through the ASIL decomposition and criticality analysis (ASIL assignment phase).
Figure 32 - Technical safety requirements definition process
4.11 ASIL allocation

The safety requirements shall be allocated to architectural elements, as early as those are available, starting from preliminary architectural assumptions and, finally, to hardware and software elements. Methods to assign the ASIL, that is an attribute of safety goals and then inherited to the safety requirements, are:

- **ASIL Decomposition**
- **Criticality Analysis**

4.11.1 ASIL Decomposition

The purpose of this safety design flow phase is to partition the ASIL requirements among subsystems/ system elements/ components. This is usually done by distributing and allocating the functional safety requirements to homogeneously or diverse redundant subsystems or components or by implementing a function with an independent separable monitoring function.

The ASIL Decomposition technique is applicable during the design phases (system, hardware and software design), and only among independents (physical separation and/or no common resources involved) system elements implementing the same safety goal. Before partitioning the ASIL, the safety channel scheme should be defined, in order to highlight the main components and/or signals that concur to determine the safety containment of the errors. Then, the decomposition approach can be performed by properly allocating the safety requirements into the elements along the safety channel and verifying the resulting level of fault containment.

Starting from the failure(s) with the maximum ASIL level, the ASIL decomposition should be applied following the decomposition schemes from ISO CD26262 (see Figure 31). Note that, each decomposition scheme requires the definition of new **additional requirements**. These are the “safety mechanisms” (i.e. filter, coherence checks, signal redundancy, etc) to guarantee the robustness of the item (the Item has to maintain the original safety integrity level), by acting as a safety barrier that guarantees the total faults containment.
In general it should be necessary to define the best criteria to select the fittest decomposition scheme; for instance, the best practice suggests an allocation of the safety barrier as close as possible to the end user, the actuator, in order to minimise the possibility of any other interference/data corruption downstream the filter.

Once a decomposition scheme has been chosen, the independence of the components after decomposition should be shown with sufficient confidence according to the ASIL before decomposition. The components are regarded as independent if the safety analysis does not find a common cause of failures (CCF) that may lead to the violation of a safety goal when the failures, each one considered individually, would not lead to violation of a safety goal.

To better understand the ASIL decomposition method, an example is described below.

4.11.1.1 ASIL Decomposition example

ITEM under safety analysis: “STEERING TORQUE OVERLAY” (STO) FUNCTION

The STO is intended to control the steering wheel in low vehicle speed condition. The STO function is based on interaction of the STOM (Steering Torque Overlay Manager) and EPS (Electric Power Steering). A torque overlay is requested from STOM to EPS in addition to the conventional steering assistance torque, by means of a CAN interface.

The Item block diagram is below:
\[ \text{STO} (\text{Steering Torque Overlay}) = \text{STOR} (\text{Steering Torque Overlay Request}) + \text{STOI} (\text{Steering Torque Overlay Interface}) \]

\[ \text{STOR} = \text{It requests a torque overlay to EPS} \]
\[ \text{STOI} = \text{It processes the STOR requests} \]

\textbf{Figure 34 - Block diagram for steering system}

\textbf{Target Function} (System functional analysis phase, see paragraph 2.1)

STO activation = Request to EPS a torque overlay on steer

\textbf{Activation Condition} (System functional analysis phase, see paragraph 2.1)

\text{Vspeed} \leq \text{threshold value (tbd)}
After performing a risk assessment (following the steps detailed in 2.3, 2.4, 2.5, 2.6, 2.7 paragraphs) the resulting situation at maximum risk shall be considered:

→ The situation at maximum level of risk is related to the malfunction “unwanted steering torque request” at medium/high speed.

**Table 4: Situation at maximum level of risk with associated Safety Goal and Safe State**

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Malfunction</th>
<th>ASIL level</th>
<th>Safety Goal</th>
<th>Safe State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medium/high speed, mu-high, straight driving, on motorway (with traffic) and suburban roads (with or without traffic)</td>
<td>unwanted steering torque request</td>
<td>D</td>
<td>Do not request any steering torque if activation conditions are not verified</td>
<td>Steering torque overlay request deactivation; fault alarm on.</td>
</tr>
</tbody>
</table>

The figure below shows the complete risk assessment row related to the situation at maximum risk of the STO function.

**Functional safety requirement:**

If $V_{speed} >$ threshold value → Torque contribution $= 0$;
Figure 35 - Risk assessment abstract – Row related to the situation at maximum risk level
Before partitioning the ASIL, the safety channel scheme shall be defined, in order to highlight the main components and/or signals that concur to determine the safety containment of the errors.

**Safety Channel definition:**

![Safety Channel Diagram](image1)

Figure 36 - Safety Channel

The safety channel scheme allows highlighting the main parts that concur to determine the safety containment of errors. The flow of the several signals involved in the process is detailed in the following Data Flow Diagram (defined in the System functional analysis phase, see paragraph 2.1):

![Data Flow Diagram](image2)

Figure 37 - Data flow diagram

The decomposition approach can be performed by properly allocating the safety requirements into the elements along the safety channel and verifying the resulting level of fault containment.

**ASIL partitioning**

The situations at maximum risk are related to the “unwanted steering torque request” failure at medium/high speed, with ASIL D level.
At first, the “ASIL D” ranking assigned to STO function is to be attributed to each element, as depicted in hereafter figure:

![Figure 38 - ASIL decomposition – first step](image)

Then, the decomposition approach can be performed by properly allocating the safety requirements into the elements along the safety channel (following the ASIL partitioning scheme suggested to ISO 26262 and reported hereafter) and verifying the resulting level of fault containment.

![Figure 39 - ASIL partitioning schemes](image)

In general it should be necessary to define the best criteria to select the fittest decomposition scheme. Regarding the STO function the following considerations can be done:

- The impact on the safety is due to a “torque overly request” erroneously sent from STOM toward the actuator through the EPS.
  
  The most natural way to avoid any consequence is to cut off the wrong command signal by means of a filter, acting as an internal safety barrier, positioned somewhere inside the function perimeter, from the source [STOM Node] up to the user [EPS Node].
convenient choice for the safety barrier allocation certainly is inside the Steering Torque Overlay Interface [STOI] sub-module, for the following reasons:

- The best practice suggests an allocation as close as possible to the end user, the actuator, in order to minimise the possibility of any other interference/data corruption downstream the filter. → STOI is the STO boundary element closest to the actuator along the command chain line.
- The safety barrier is requested to validate the coherence of the steering torque overlay request with the activation conditions expected for STO activation. The EPS, in which STOI is embedded, is in charge of controlling the vehicle speed, so that it already manages the same signals needed for the filtering action.
- EPS (Normal Production version) is to be considered a strengthened product by now, with a high safety integrity level guaranteed by the supplier.

As a consequence of the above consideration, since the safety barrier guarantees the total fault containment, both STOM node and C-CAN can be considered at minimum risk, so it is reasonable to assign them the lowest safety integrity level, by decreasing from D to QM ranking.

In conclusion the more convenience choice (orange highlighted in Figure 39) is the following

STOM → QM    EPS → ASIL D    + additional requirements

Figure 40 - ASIL decomposition

**Additional requirements definition**

To apply the above ASIL partitioning, the independence of the components after decomposition shall be shown with sufficient confidence according to the ASIL before decomposition. The components are regarded as independent if the safety analysis does not find a common cause of
failures that may lead to the violation of a safety goal when the failures, each one considered individually, would not lead to violation of a safety goal. The functional independence comes from the availability of different and independent control signals to cross check by the STOM-EPS in order to validate the “Steering Torque Overlay”.

**Requirement:** the speed level shall be acquired, validated and diagnosed by EPS cross-checking the following “Braking Node” different and independent signals:

- MSG3.LHR/RHRFastPulseCounter
- MSG1.VehicleSpeedSignal

Both these signals come from the same “Braking Node”, therefore the cross-check on speed level is meaningful only if it is guaranteed the absence of any CCF (common Cause of Failure) inside the “Braking Node”, and a particular grade of robustness for the MSG3 signal (i.e. CRC and rolling counting techniques).

### 4.11.2 Criticality Analysis

A second technique to ASIL assignment is the *Criticality analysis*.

*Criticality analysis is an analytical method which to be applied to architectural elements and the already assigned ASILs of the elements* (ISO CD26262 – Part 9).

The criticality analysis allows investigating the possibility of reducing the ASIL inherited by some internal elements or components of an item or element. The criticality of each element defines the potential of this element to violate a safety goal. Each elements of the item, implementing at least one safety goal, shall be ranked in one of the following “criticality class” (ISO CD26262-part 9):

- **CC3 – Safety critical**
  Denotes a function, where a single deviation from the specified function may cause a violation of a safety goal

- **CC2 – Interference free**
  Denotes a function, where a single deviation from the specified function cannot cause a violation of a safety goal, but the combination with a second failure of another component may cause a violation of a safety goal.

- **CC1 – Interference free**
  Denotes a function, which has not critically category CC2 or CC3 and does not influence the safety relevant behavior of any CC2 or CC3 function, either by direct interface or through a shared resource or through an intermediate element.
The resulting ASIL shall be derived from the criticality level estimated for each element:

<table>
<thead>
<tr>
<th>Initial ASIL of the element</th>
<th>Criticality of the element</th>
<th>Resulting ASIL:</th>
<th>Resulting ASIL:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIL A</td>
<td>CC1 QM</td>
<td>ASIL A</td>
<td>ASIL A</td>
</tr>
<tr>
<td>ASIL B</td>
<td>CC2 QM</td>
<td>ASIL A</td>
<td>ASIL B</td>
</tr>
<tr>
<td>ASIL C</td>
<td>CC3 QM</td>
<td>ASIL B</td>
<td>ASIL B</td>
</tr>
<tr>
<td>ASIL D</td>
<td>QM</td>
<td>ASIL C</td>
<td>ASIL D</td>
</tr>
</tbody>
</table>

### 4.12 Current EAST-ADL language support for safety workflow

The safety-related information is related various parts of the EAST-ADL system model, and this section will propose how and where to represent the relevant concepts.

#### 4.12.1 Identification of Safety-related information

System safety activities according to ISO26262 rely on various information. *Error! Reference source not found.* illustrates some of these. Below is a list of information that is relevant for safety, and a discussion how it can be represented in EAST-ADL. The list is not systematic, and it is based on ISO26262 and Figure 41.
Item
Item is the system or array of systems or functions that are subject to safety assessment. This means that it is not possible to have a strict definition of what Item is in terms of modelling elements. Rather, Item may be a set of modelling entities on several levels of abstraction.

Hazards
A hazard is an event which might potentially cause harm. These are identified independently of the system solution and can thus be associated with the feature on vehicle level.

Safety Goal
The safety goal is the inverse of a hazard, i.e. the goal is to avoid $H$ to happen, where $H$ is the hazard. Safety goal is a specialization of ADLRequirement and should refer to the Hazard that it addresses.

ASIL identification and allocation
An ASIL level is defined for each hazard, based on its characteristics: Severity, Exposure and Controllability of the Hazard event. This is captured as a hazardClassification of the Hazard entity. When allocateing ASIL levels to functions and components, the SafetyRequirement is used. SafetyRequirement has ASIL attribute.

ASIL decomposition
When doing ASIL decomposition, it is necessary to refer to the initial ASIL that is decomposed. This is supported by the use of DeriveReqt constructs: For example, the ASIL A and C requirements of two realizing components will be derived from the ASIL D.
requirement of their composite component, or the ASIL D requirement of the abstract component that they realize. In case there is no common entity to refer to, the ASIL D requirement would have to be allocated to each of the components, and a rationale provided why it can be derived to an ASIL D requirement on the same component.

- **Safety Requirements**
  Safety Requirements are preconditions or abilities that an item shall fulfill to achieve the safety goals. It is thus misleading to denote the general ADLRequirement specialization with ASIL “Safety Requirements”. Rather, this is an ASILRequirement. Safety Requirements are a broader category of requirements and includes both quality and functional requiremnts. They are probably best identified through the Functional ans Technical Safety Concept containers.

- **Functional Safety Requirements**
  See Functional Safety Concept

- **Technical Safety Requirements**
  See Technical Safety Concept

- **Functional Safety Concept**
  Functional Safety Concept is a specification of the requirements on the basic functionalities and their interaction necessary for achieving the safety goals. It is thus a set of requirements and can be collected by a RequirementsContainer specialization. The contained requireemnts are Functional Safety Requirements.

- **Technical Safety Concept**
  Technical Safety Concept is the set of technical safety requirements needed to implement and partition the functional safety concept on the system architecture. The contained requirements are Technical Safety Requirements.

- **Errors and error events**
  An error event is a propagated error which may be system internal or external. External error events may propagate and cause Hazards. This is modelled with a relation HazardCause, which identifies which hazards result from each manifest error, i.e. an error event on the final ErrorPort

- **External Measures**
  Requirements on external measures may prescribe conditions outside the EE architecture necessary to meet the safety goal, or assumed to be met in order for the chosen safety concept to be sufficient to meet the safety goal..

- **System Boundary Definition**
  System Boundary Definition represent the Item’s boundary and interfaces with other items or elements. Given the loose definition of Item, this cannot be a strict definition in the EAST-ADL, but rather a set of constructs. On Analysis level, the boundary definition may be a composite ADLFunction with its interfaces or several ADLFunctions. On design level there would be two or three parts, ADLFunction(s) in FDA, HW entities in HDA and ADLFunction(s) in MWA (middleware). On implementation level, there would again be three parts, one or several SW components, one or several HW entities, one or several BSW components.

- **System Functional Analysis**
  System Functional Analysis is there to understand the system and how it may fail. It could be based on System Function Identification, Use Cases, Data Flow Diagram Definition and Block Diagram Definition.
The language support in EAST-ADL is designed to demonstrate how it is possible to trace the safety requirements while also giving evidence that single points of failure are managed.
5 Conclusions

In this document, we have reviewed a number of model-based safety analysis and fault diagnosis techniques. We have also examined the safety design flow as set out in ISO 26262. Now it is time to consider how this information can be used to help inform the development of EAST-ADL.

5.1 Safety Analysis of System Designs

As explained in Section 2, safety analysis techniques fall generally into two categories: compositional techniques and behavioural techniques. Both categories have their own advantages and disadvantages: behavioural techniques are normally more highly automated, able to construct models and analyse them from formal specifications of the system without additional human input, but compositional techniques tend to be less computationally expensive than behavioural techniques, which tend to rely on model checkers and thus suffer from state-space explosion. This increased efficiency allows such techniques to include greater capabilities; in the case of HiP-HOPS, for example, this latitude makes it possible to develop extensions into reusable patterns of failure, temporal failure logic, and automatic design optimisation of dependability versus cost. HiP-HOPS also supports both FTA and FMEA but produces them via a deductive process, avoiding the combinatorial explosion issues inherent in purely inductive techniques like DCCA or Altarica. In general, compositional safety analysis techniques are more flexible and can be applied to a greater range of systems, because they do not always rely on formal specifications and the potential domain-specific conflicts that they can introduce.

Both behavioural methods like Altarica, FSAP/NuSMV-SA, and DCCA as well as compositional techniques like CFTs and HiP-HOPS have well-established tool support, often linking with commercially available modelling tools such as Matlab Simulink. The NuSMV model-checker tool is widely used not only in safety analysis but also in other areas such as fault diagnosis. One of the chief differences in the tools is their performance; tools for compositional techniques tend to be faster for larger systems, though for simpler models the differences are much smaller.

Of the compositional techniques, FPTN, CFTs, and HiP-HOPS are all similar in function, relying on component annotations to model the propagation of faults through a system, but HiP-HOPS has the greater capabilities owing to its optimisation and temporal extensions. FTPC behaves in a similar manner but is designed for the real-time software domain and uses a more specialised notation. This makes it well suited for that domain but means it would require modifications to be applicable to more general types of systems. SEFTs provides another state-based notation that is suitable for use with more dynamic systems and thus offers capabilities similar to HiP-HOPS and Pandora; however, its heavily modified fault trees means it can no longer be analysed using traditional means and other techniques such as Petri Nets must be used. This increases the complexity of the technique and requires more tools (and more time) to perform an analysis. The AADL error annex is a powerful and expressive approach, offering state-based semantics in addition to fault propagation capabilities. However, it too requires specialised analysis techniques; it is not possible to directly convert state-based semantics to fault trees (the most common means of analysis) without some additional method of handling the dynamic and temporal information it implies, as SEFTs and Pandora demonstrate.

Of the behavioural techniques, Altarica provides a similar notation to that of the AADL error annex – a text-based, state-transition notation that can be used by a model checker to verify that a system meets its safety and reliability requirements. It also enables analysis using several different methods, e.g. fault trees, Petri nets, Markov chains etc. However, Altarica does have a number of drawbacks, particularly the difficulty of modelling fault propagation and its vulnerability to state-space explosion for larger systems. In addition, conversion to other techniques (e.g. fault trees) is heavily dependent upon the dynamic or static nature of the system under consideration. FSAP/NuSMV-SA has both similar benefits and similar drawbacks; it uses the popular NuSMV model checker as its engine, which is very well established, but the fault trees it produces are very...
flat and do not show propagation of failures through the system, nor do they cope well with ordering information, and like Altarica it can suffer from performance problems if there are many states in the system. DCCA is a newer approach that tries to overcome some of the problems experienced by Altarica and FSAP, but it requires relatively complex CTL-based formalisations of the system and its effectiveness still varies depending on the number of states involved. It does however provide capabilities to model dynamic situations by means of its Deductive Failure Order Analysis.

Overall, HiP-HOPS offers the most capabilities and, since its failure semantics are independent of the modelling language used, it is flexible enough to be incorporated into other modelling languages. It could therefore provide a basis for defining error modelling and safety analysis in EAST-ADL, and its multi-objective optimisation capabilities could be used to support optimisation in EAST-ADL as well. However, HiP-HOPS is based on fault propagation and lacks the concept of states. Although this is not necessarily a problem in terms of safety analysis, as Pandora makes it possible to extend HiP-HOPS with temporal logic to implicitly represent transitions to failed states, explicit state modelling could still be useful for formal verification purposes. For this reason, the state-based error model annex of AADL could also prove useful as an input to the error modelling in EAST-ADL and we believe that it would still be possible to harmonise a state-based error model with HiP-HOPS and thereby continue to derive the benefit of its analysis & optimisation capabilities.

5.2 Online Fault Monitoring & Diagnosis

Online Fault Diagnosis uses many similar techniques to offline safety analysis, e.g. fault trees, but they differ in how these techniques are used; instead of trying to reach conclusions about how safe or reliable a system is, instead they are used to determine the causes of a fault that has already occurred. They fall into three main categories: rule-based, model-based, and data-based approaches.

Rule-based approaches make use of knowledge bases about the system, frequently containing inference rules that link causes to effects. When patterns of behaviour match rules, these rules can be followed to determine either the cause (by going backwards) or the consequence (by going forwards). Modern rule-based systems tend to make use of more advanced approaches such as neural networks, which learn the behaviour of the system. However, rule-based approaches have a number of drawbacks – in particular, they struggle with temporal reasoning. It is also difficult for them to distinguish between faults with similar symptoms, or if a system experiences multiple faults at once. The biggest problem however is the number of rules required to describe the behaviour of the system; for larger systems, the knowledge base grows to a prohibitive size.

Model-based approaches overcome the problem of maintaining huge knowledge bases by representing the knowledge as part of the system model itself. They fall into two categories: those that model the failure behaviour of the system, in which case diagnosis takes place by monitoring for certain symptoms and which can then be backtracked to determine the root causes, and those that model the nominal behaviour of the system, in which case diagnosis takes place by comparing the expected behaviour of the system (as predicted by the model) against the actual, observed behaviour of the system. The failure-behaviour type includes fault propagation models, including fault trees and cause-consequence diagrams, which can be effective but require special handling to model dynamic situations, as well as causal process graphs like digraphs, which have been extended with a number of additional capabilities but do not scale well for large systems, and finally functional decomposition techniques like GTSTs, which are very efficient and thus suitable for real-time systems but which tend to be primarily qualitative approaches.

Approaches that model the nominal behaviour include both behavioural simulation approaches like the GDE and qualitative simulation approaches like 105. The former tend to suffer from problems when trying to represent dynamic, multi-phase systems but are capable of diagnosing multiple simultaneous faults, which is a powerful capability. The latter type are very powerful as well,
offering a number of capabilities including the ability to handle states and can also facilitate other types of analysis including risk analysis with FMEA. The fact that they model nominal behaviour means they can handle unanticipated (and thus un-modelled) faults but also means that it is more complex for them to diagnose problems that are anticipated when compared to failure-behaviour approaches.

Finally, data-based approaches study the data provided by the system to determine patterns, comparing these patterns of behaviour against the expected behaviour of the system. They include a number of different techniques depending on what is measured (e.g. single variables versus multiple variables) and how (e.g. neural networks). These types of approaches can be very effective but are heavily dependent upon the type and quality of data provided by the system monitors; as a consequence, they are less generally applicable than some of the other types of approaches.

For use with EAST-ADL it would seem logical to focus attention primarily on the model-based approaches to fault diagnosis, as EAST-ADL is a modelling language. EAST-ADL offers both nominal and error modelling and so should be able to support both types of techniques. If fault trees can be generated from EAST-ADL error models, then this would provide a potential means to conduct fault diagnosis on the basis of the failure-behaviour of the system; alternatively, it may be possible to derive other types of models, e.g. CCDs or GTSTs. Alternatively, simluation-based approaches that use information about the nominal model may be able to support comparative diagnosis by comparing this simulated behaviour against the actual behaviour of the system.

5.3 Safety Design Flow

The safety design flow traces the guidelines to perform a safety analysis of E/E systems compliant with ISO CD26262. The safety design flow activities are focused to concept phase and design phase of the safety lifecycle.

To perform the item safety analysis, it is essential to properly understand the item itself in terms of input(s)/output(s), functionality, interfaces, environmental conditions and, to define the item target function, which will be the base for the subsequent malfunctions definition phase. Before the safety analysis activities are performed, the boundary of the item and the item’s interfaces with other elements should also be determined.

To evaluate the risk associated with the item under safety analysis, a risk assessment should be carried out, starting from the scenarios definition (operative conditions & environmental conditions), in respect of the item functionality under analysis. To identify the hazards it is essential to define the malfunction(s), the misuse(s) and eventual maintenance condition(s) related to the item. If the item target function(s), that is the function description in terms of output(s) behaviour, has been correctly identified and described, the malfunction can be always defined in terms of anomalies of function activation. Therefore the hazards have to be identified by analysing the malfunctions in concomitance with a particular scenario’s conditions.

For each potential hazardous situation, the severity, controllability and exposure values should be ranked, to determine the associated Automotive Safety Integrity Level (level of risk). It is important to remark that the controllability levels assigned to the various situations should be assessed through specific testing on the road, fault injection, etc.

For each safety relevant situation (ASIL ≠ QM) a safety goal and, if applicable, a safe state should be defined. Starting from the safety goals, the functional safety requirements should be defined, taking also into account the System FMEA and FTA results.

The functional safety requirements should be detailed in terms of technical safety requirements, which should be defined during architecture and system design. The process of technical safety requirements definition is an iterative process that passes through the ASIL decomposition and criticality analysis (ASIL assignment phase).
5.4 Final Summary

This document provides a wide-ranging review of safety analysis, fault diagnosis, and safety workflow topics. It is hoped that this information can be used to help inform decisions about the future design and direction of the EAST-ADL language in terms of its support for safety and dependability concepts. In particular, the language should make it possible to use a safety design flow compatible with that defined by the upcoming ISO 26262 standard, as described in Section 4, including support for topics such as hazards, safety goals and requirements, and the representation and decomposition of ASILS. Many of these concepts are already represented, as mentioned at the end of Section 4, but there may still be further development necessary, e.g. in the area of ASIL decomposition.

In addition, in order to support this type of design flow, EAST-ADL needs to support safety analysis, e.g. using well-established methods such as FTA and FMEA. As described in Section 2, there are many different techniques and even paradigms of model-based safety analysis and EAST-ADL may benefit from adopting some of the concepts found in these techniques, e.g. fault propagation as in HiP-HOPS or state-based failure semantics along the lines of AADL. Ultimately, the safety analysis techniques supported by EAST-ADL may be extended further in the future to also support fault diagnosis and error handling techniques such as those described in Section 3.

Finally, it is hoped that this document can evolve over time into – or serve as the first step towards – a more comprehensive proposal for the additions or changes necessary in EAST-ADL to provide full and effective support for the design and analysis of safer automotive systems.
References


Calabria, Rende, Italy, 23-24.11.2006, 2nd workshop on networked control system and fault tolerant control


