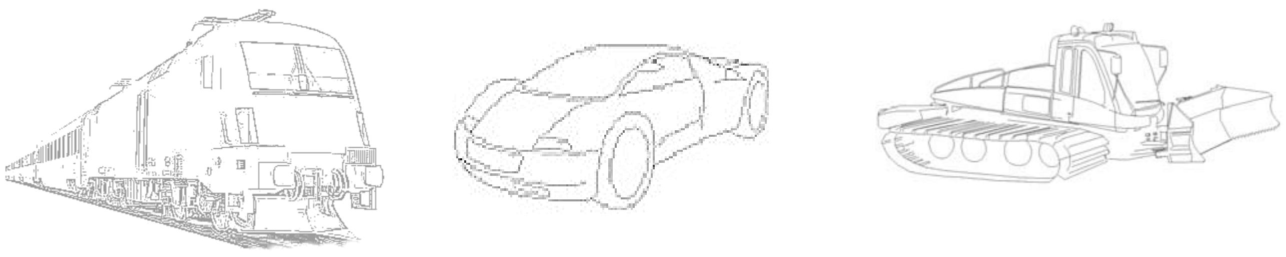
MOGENTES

Model-Based Generation of Test-Cases for Embedded Systems

Fault Models

(final version)



Project	MOGENTES		**Contract Number**	216679	
Document Id	3-09_D3.1b_1.0r	**Date**	2009-12-29	**Deliverable**	D3.1b
Contact Person	Georg Weissenbacher		**Organisation**	ETH Zurich	
Phone	+41 44 63 27970		**E-Mail**	georg.weissenbacher@inf.ethz.ch	

Distribution Table

Name	Company	Department	No. of copies	Hardcopy/ Softcopy
All MOGENTES team members			1	SC

Change History

Version	Date	Reason for Change	Pages Affected
0.1w	Dec 8, 2009	First draft, based on 3.1a, integrated comments from AIT, FFA, and TUG.	All
0.2w	Dec 17, 2009	Restructured document. Integrated new version of Section 6 from BME, new version of Section 5.2 from SP, rewrote parts of Section 4 and added new material to Section 5.1.1.	All
0.3w	Dec 24, 2009	Inputs and changes by AIT, Prolan, RELAB, SP, Thales. Section 3 is significantly extended, a conclusion was added.	All
1.0r	Dec 29, 2009	Added contributions paragraph, updated version	All

Contents

Figures

Tables

1 Introduction

A main objective of the MOGENTES project is to support the generation of suites of test cases for the industrial demonstrators of the project that exercise an implementation to the degree required by the relevant safety standards (e.g., EN 50128/129, ISO 11783, ISO 26262 and IEC 61508). The fact that the actual implementation of a system consists of hardware components as well as software modules complicates this attempt. Current verification and risk analysis techniques address these two aspects of the system in a very different manner. While software is verified by means of exhaustive testing (where “exhaustive” is typically defined by means of structural coverage metrics such as Modified Condition/Decision Coverage, MC/DC), the safety and reliability of the hardware components is established using methods like Fault Tree Analysis (FTA), or Failure Mode and Effect Analysis (FMEA), or fault injection.

The Model-Driven approach underlying the MOGENTES project allows us to analyse the system as a whole, since the system model describes the hardware components as well as the software modules in terms of a uniform modelling language. In order to be able to devise a holistic approach, the test case generation technique has to be based on a common coverage objective. The solution we propose is based on the observation that *mutation testing* and *fault injection*, both common techniques in software and hardware verification respectively, rely on the deliberate introduction of errors into the system under test and on analysing the behaviour of the interfered system. We propose to assess the quality of the test suite by measuring the number of mutations and injected faults it is able to detect. Furthermore, we prefer small test suites, i.e., a test suite that achieves a certain coverage with less test cases is more efficient than a larger test suite achieving the same coverage.

Mutations and faults are two very different concepts, though. Mutations are small, syntactical modifications of the source code of a software module, a state diagram of a subsystem (e.g. of a transition guard), or similar model modifications. The fact that a test suite is able to detect mutations is mostly an indicator for an adequate structural coverage of the code. The mutation operators applied to the program do not necessarily have their origin in typical programmer mistakes (though this is a possibility), but are created with the incentive to force the creation of an adequate test suite. While mutations are used to remove faults (bugs) from software before runtime, fault injection is used to evaluate the effect and fault handling of faults occurring during run-time. In other words, mutations are used for fault removal and fault injection for fault tolerance. Fault injection traditionally aims at simulating hardware failures by means of mimicking (realistic) causes of failures to evaluate fault tolerance, for instance by examining the error detection coverage. Dependable embedded systems need to be robust with respect to *soft errors* occurring in the computer hardware. Soft errors are caused by transient faults that alter the binary values stored in latches, flipflops and other state elements without causing any permanent damage to the hardware. Thus, a soft error consists of one or more bit-flips in registers, memory words or latches caused by a single transient fault. Important sources of soft errors are ionizing particles such as heavy-ions, alpha particles and high energy neutrons, as well as electrical interference generated internally on the chip because of electrical coupling and power supply noise. While faults caused by electrical interference could be avoided by careful circuit design, it is nearly impossible to avoid soft errors caused by ionizing particles. In fact, such errors are expected to become the dominating source of hardware related failures in future digital circuits [Baumann, 2004]. The effects of soft errors can be simulated in e.g. behavioural models or software by using the failure mode functions presented in Section 4.2.

Fault injection mechanisms can be implemented directly into the model, by using additional software routines, or by adding extra hardware. The respective approaches are called *model-implemented fault injection* (MIFI), *software implemented fault injection* (SWIFI), and *hardware implemented fault injection* (HIFI). In fault injection, the relationship between faults, errors, and the resulting failure effects is more intricate than it is the case with software mutations. A fault results in the occurrence of an error, which in turn may manifests itself in an observable failure [Laprie, 1985], [Avizienis, 2004]. A failure of a component at one level of abstraction may cause a fault in the level above which in turn may cause a new failure at this level of abstraction (thus forming a causality chain). For instance, a hardware component failure may result in the delay in the transmission of TCP/IP packages, i.e. result in a communication failure. The Model-Driven approach allows us to implement the communication failure by means of using MIFI. Even though, in our example, the delay is *caused* by a hardware failure, the communication failure itself is strictly seen not a hardware failure (it may result from different causes, e.g., by increased traffic on the network, a synchronisation failure or a software bug). The model of the failure modes and effects has to be as truthful as possible to the failure modes that may occur in the actual implementation.

The aim of this document is to provide an overview of the faults and mutations relevant to the MOGENTES project. Instead of providing an exhaustive catalogue of all conceivable failures, we restrict the scope of this document to the needs of the domain-specific demonstrators.

1.1 Purpose and Scope

This document provides a taxonomy and catalogue of mutations and failure modes and effects. The main focus is on the domain-specific faults and mutations relevant for the MOGENTES demonstrators; the precise connection between the demonstrators and the mutations and failure modes is given in Section 3. Section 4 presents our taxonomy of mutations and failure modes, followed by the mutation/failure mode catalogue in Section 5. Section 6 introduces an additional aspect to the library of fault patterns/templates: it focuses on the *qualitative modelling* of faults, errors, and failures. Spatial and temporal compaction techniques are proposed to master the *model complexity problem* occurring during the analysis of faults effects (that is required for test optimization). The presentation is at a conceptual level that does not restrict the modelling languages to be used in the project.

While Section 6 deals with fault modelling aspects, the actual formal analysis of the model and the injected faults, which underlies our test case generation approach, is discussed in further deliverables such as D4.1 and D4.4 and not in the scope of the deliverable D3.1.

This document is an update of D3.1a, taking recommendations from the first annual project review into consideration, as well as further experiences and inputs gathered during the progress of the project.

1.2 Individual Contributions by the Project Partners

Compared to D3.1a, the following major contributions were made to the deliverable:

- Section 3 was extended by AIT, Prolan, RELAB, SP, Thales.
- Section 4 was updated by ETH, SP.
- Section 5.1.1 was rewritten by ETH.
- Section 5.1.2 was rewritten by AIT.
- Section 5.2 was extended by SP.
- Section 6 was updated and extended by BME.
- Section 7 was added by AIT, SP.

2 Background

To motivate the taxonomy and the catalogue of mutation and failure behaviour provided in Section 4, we provide a brief overview of mutation testing and fault injection.

2.1 Mutation Testing

The idea of mutation testing is to introduce faults in either the specification or implementation and then generate test cases that discriminate specified from mutated behaviour. Mutation testing is based on the **coupling effect** [DeMillo, 1978] and the **competent programmer hypothesis** [Acree, 1979]. The former hypothesis states that test cases that can detect simple faults are likely to find more complex faults. The latter states that programs are mostly correct. Programmers often make common mistakes like misnamed variables or wrong conditions in branch statements. Mutation testing uses these assumptions to form fault models comprising a set of mutation operators. Mutation operators for specifications are analysed by Black et al. [Black, 2000]. The examples in [Black, 2000] are modelled in SMV [McMillan, 1992], the language for the SMV model checker.

Originally mutation testing considered faults introduced in the implementation [DeMillo, 1978], [Acree, 1979].

Specification mutation was initially proposed by Amman et al. [Ammann, 1999]. In this context, mutation testing deals with introducing faults in the model or in the temporal requirement specification. In the first case, every property violation of the model leads to a counterexample. These counterexamples are **negative** test cases as they contain behaviour that must not be implemented by the SUT. In the second case temporal properties are mutated and produce counterexamples as **positive** test cases.

2.2 Fault Injection

Hardware faults resulting in errors may be detected and recovered by error detection and error recovery mechanisms. Validation of error detection and error recovery mechanisms is an important but challenging task. One way to validate such mechanisms is to accelerate the occurrences of faults in the system by means of *fault injection* [Iyer, 1995].

Fault injection denotes the deliberate introduction of faults into a SUT. As fault injection (also known as fault insertion testing) has become widely used as an experimental dependability validation method, many different techniques for injecting faults have been developed. Fault injection accelerates the occurrences of faults in a system and the main purpose is to evaluate and debug error handling mechanisms. It is used at various abstraction levels and phases of the development process. Fault injection is e.g. mandatory in safety standard IEC 61508 (adapted by the automotive industry as ISO WD 26262) when claimed diagnosis coverage is at least 90%.

Fault injection is traditionally used for emulating hardware faults, where different techniques normally are divided into *simulation-based* and *physical* techniques depending on whether faults are injected into hardware models (e.g. VHDL models), or into an actual physical system or prototype.

To avoid focusing only on the target for fault injection, another classification is instead based on *how* fault injection mechanisms are implemented. Thus, the focus is on whether extra hardware are used for fault injection (denoted as *hardware-implemented fault injection or HIFI*) [Madeira, 1994], [Karlsson, 1994], [Vinter, 2005], or if extra software is used (*software-implemented fault injection, SWIFI*) [Han, 1995], [Carreira, 1998], [Martins, 2000], or if fault injection mechanisms are added directly into models of hardware, models of software or models of systems (denoted in this deliverable as *model-implemented fault injection, MIFI*) [Kumar, 1997], [ISAAC, 2007], [Vinter, 2007].

3 Domain-specific Examples for Fault Models

This chapter collects experiences with design and implementation faults from the industrial demonstrator partners. The motivation behind is coupled with the motivation for mutation-based test case generation, which is based on following concept:

– "Competent programmer" assumption: The implementation of a given specification will be mostly correct. Hence, if there are bugs in the code then mostly as small deviations from specifications.

– To find these, introduce fault models (mutation operators) that simulate these kinds of "misunderstandings".

– Mutate the original specifications (which are assumed correct) with these operators and try to create a test case that tells you whether the implementation follows the correct specifications or the mutated one.

Since for any specification the set of potential mutations quickly tends to become huge (number of mutation types * number of possible locations), it is crucial to identify means which allow for significant reduction of this variability. One such instrument is described in chapter 6, the other is to exploit experiences of domain experts, in order to be able to identify relevant mutations (and locations).

3.1 Railway Domain: Fault Model Considerations related to ELEKTRA

Several connotations of the term "fault" need to be distinguished:

1. *Software faults* that a programmer typically makes when implementing a given set of requirements. The goal of the test cases generated and performed is to detect and subsequently eliminate as many software faults as possible.
2. *Faults in the environment* of the interlocking system which the system has to handle. They are therefore part of the input space of the interlocking system and reflected in the model of each field element. They can be simulated ("injected") with TRSS's test environment and simulator ("EPS"). Because environment faults are part of the model, test cases against the set of requirements can be generated partly without (in practice: a small subset of all test cases) and partly with (in practice: the majority of test cases) the presence of environment faults.
3. *Internal (system-internal) hardware faults*. These must be detected and lead to subsystem or system shutdown and subsequent reboot. The detection methods are: CRC checks, input and output voting (taking advantage of the 2-channel architecture), online tests of CPU and RAM. (Note that these measures also protect against (remaining) software faults.) These faults are not of concern to system test and, consequently, not part of the MOGENTES project.

3.1.1 Possible Sources for Software Faults

In MOGENTES, UML/OCL has been chosen as languages for the test models from which test cases are derived. It should be noted that mutations in UML state diagram do not directly reflect implementations faults, because the used programming languages (Chill and PAMELA) imply other coding faults at syntax level. However, on the (qualitative) abstraction level of UML, the following relationships between developer mistakes and mutations at UML model level are seen, which are considered to show a good correspondence with qualitatively equivalent design and implementation faults:

Mutation	Interpretation, Comment	Probability
Delete a guard statement on a transition	Design fault: missing implementation of a requirement or a part of it, typically a condition	fairly high
Delete a state transition	Design fault: missing implementation of a requirement or a part of it; typically treatment of a special case	fairly high
Delete a state	Design fault: missing implementation of a requirement or a part of it; requires also deletion of all associated transitions	rather low
Insert a state transition	Additional behaviour implemented	rather low

Choose another initial state	Initialisation fault	fairly high
Exchange start state of a transition	Requirement misinterpretation: hard to interpret at programming language level; could correspond to a misinterpretation of a requirement, e.g. the condition for leaving a state sequence too early or too late.	low
Exchange end state of a transition	Requirement misinterpretation: similar as before, but could e.g. correspond to confusing follow up activities after a certain event	rather low
Permute start and end state of a transition	Implementation fault; corresponds to inverting the logic flow (e.g. from error state to correct state when an error occurs)	low
Change operator in a guard statement	Implementation fault: depends on type of operator. E.g., ‘>’ → ‘=’ is rather unlikely than ‘>’ → ‘>=’.	rather low - high
Invert result of a Boolean guard statement	Implementation fault: depends on complexity of requirement	rather low – fairly high
Replace variable in a guard statement with constant	Implementation fault: can happen when progressing from older SW versions	fairly high
Use different variable (of same type) in a guard statement	Implementation fault: depends on similarity of variables	fairly high
Delete trigger from a transition	Implementation fault, corresponds to guard deletion	fairly high
Replace trigger with another trigger (of the same type, if applicable)	Implementation fault, e.g. mixing up inputs or input events	rather low – fairly high
Change data of a trigger	Implementation fault, only for internal triggers (triggers which are inputs are under the control of the test environment). Problem: internal triggers of test model need not to correspond to internal control of implementation	(little relevance)

Table 3-1: Correspondence between mutations in UML state diagrams and implementer mistakes

(The distinction between design and implementation is related somewhat to the complexity of the underlying mistake; they shouldn’t be regarded as strict.)

Notes:

- Probabilities in Figure 3-1 should only be considered as hints; log records are not too indicative in that direction.
- Misinterpretation of requirements usually affects more than one element at test model level.
- A large part of software faults detected during system integration or system test is related to some boundary situation, e.g. a state change in the first or in the last element of a train route, a state change in the element just before a train route, or a state change of an element which is at the boundary of a specific interlocking system’s area of control responsibility (does not apply to the MOGENTES example/demonstrator). A typical symptom is that channel A and channel B produce different outputs. Only a very detailed analysis of defect data could show which percentage of such faults is due to the requirements not being detailed enough and which percentage is due to errors in requirements interpretation and implementation.
- Another important part of software faults stems from the fact that the ELEKTRA software system is typically extended, i.e. functionality or new elements are added. Then sometimes a special case is not implemented in a hitherto generic algorithm, e.g. message handling for a new element. Thus, a requirement is only partially implemented.

- In addition to the generic mutations at model element type level as shown in the table above, specific "confusions" at object level could also be of interest. So, for instance, confusing 'left' and 'right' of a switch element is somewhat more likely than to use a wrong value for a speed limit (which could be considered as unlikely). However, it is difficult to get sufficiently significant empirical data for basing preferences on them.

3.1.2 Faults of Interlock Periphery

For pin-based discrete inputs (scans) the hardware fault model applies (refer also to D1.1.b). The models of the field elements contain the information which element states the control logic is able to discern and how these element states can be reached through simulator commands. For example, the information provided by a track supervision element (i.e. whether a track is free or occupied) may become inconsistent. The input may become inconsistent spontaneously or as a result of issuing a command for an element. The respective hardware fault can be modelled using the Boolean FMFs in Section 5.2.1.

For message based inputs, the communication fault model applies (see 5.2.5). Since communication with another interlocking (or some other) system is not part of the MOGENTES set of requirements, this does not apply.

3.2 Railway Domain: Consideration of Faults Related to the Objs module of ELPULT

With respect to the development of the ELPULT system, Prolan distinguished two kinds of knowledge of faults on model level: knowledge of railway domain, and knowledge of process control domain. Both are described in the following sections.

3.2.1 Faults related to the Railway Domain

3.2.1.1 Parameterization

The generic railway application is used for a specific station by parameterization. The information of railway objects is centralized to the safety equipment (interlocking system). The number and type of the objects are different on the specific stations. The generic code is parameterized to each application, leading to following potential faults:

- Object identification of input or output RTU signals. (Is the physical signal bound to the correct database record?)
- Is the signal connected to the proper bit of the railway object?
- Missing RTU signal?
- Is the input connected to a proper RTU?

3.2.1.2 Redundant Information

The input contains redundant bits to improve safety level. Two solutions are used:

- *Valent – antivalent:* simple redundancy with inverted signals. Some information is represented redundantly inside the relay based interlocking system: e.g.: positive or negative control of points.
- *Natural Valent – antivalent:* based on the interlocking logic. Two different contacts of a relay are used, one is opened and the other is closed in a state.

The Objs module compares this redundant data, and if there is an inconsistency after a predefined time window, the quality (status) of the data is set to invalid.

3.2.2 Faults related to Process control

3.2.2.1 Communication

The used protocol conforms to IEC 60870-5-104. Fault models are according to the EN 50129-1 standard and described in Section 5.2.5.

The quality of received data is based on the following types of information:

- Quality of data (e.g., NOCONNECTION, INVALID)
- Time of last change
- Identification of data (The Objs module receive series of input points, which are identified using this field.)

Whenever a value of a input point is used in a processing algorithm, the quality-related fields are also filled or tested. The fields are stored along with the data in the database written by the Objs module.

3.2.2.2 Considerations about Fault Injection

Elpult uses the 2002 safety architecture, i.e. 2 identical channels, both of which must provide the same information for being considered correct. The following error checking operations (which are potentially subject to fault injection) are realised:

- Detected by single process (standalone). For example: inconsistent bits.
- Comparison of outputs in two channels
- Comparison of parameters (static data check)
- Missing control

The following fault categories should be detected by means of comparison of the two output channels in 2oo2:

- Data points are different over the time window
- Short transients can appear in different sequence
- Missing data in one channel
- Missing data in both channels (life-signal’s time-stamp is considered)
- Both channels are faulty

3.3 Automotive Domain: Fault Models for Steering Anti-Catchup (SAC)

FFA does currently not use domain specific fault models, but relies on the generic fault models described in Section 0. Therefore, this section provides a brief description of the SAC testing setup, followed by a discussion of possible domain specific fault models and mutations which may be considered in the MOGENTES project.

3.3.1 Integration of the SAC in the Software Environment

The SAC function is embedded in a set of algorithms that realise either further features (e.g. VGR) or have control and safety tasks. For safety reasons, the set is designed as a "horseshoe". Hereby an input firewall (IFW) protects the inner functions from faulty signals, while an output firewall (OFW) checks the control signals before leaving the set (Figure 1-1). This implements that the inner blocks do not have an own firewall.

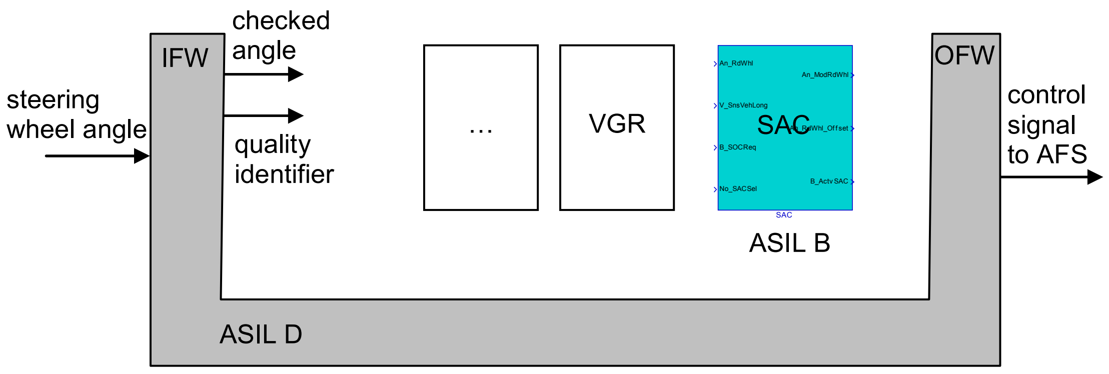


Figure 3-1: Integration of the SAC in the Software Environment

Introducing a horseshoe architecture allows to apply different levels of safety to the single components. The levels are classified with regard to the Automotive Safety Integrity Level (ASIL). ASIL ranges from level A, where low safety is necessary, to level D, requiring the highest safety. With the horseshoe satisfying the highest safety level D, the internal algorithms only need to be of level B. Many safety queries are shifted to the IFW of the outer algorithm.

Another aspect is the use of an identifier that characterise the quality of a signal (Quality factor = QF). Hereby properties of the signal are encoded as integer numbers. Zero means that a signal always is correct. The quality factor is sent with the signal, helping subsequent algorithms in judging its reliability.

The embedment of the algorithm in a set and in the software environment has also another effect. The input signals are no longer floating point numbers, but unsigned integers[1]. The conversion from integer to float is done in the firewall.

In final software versions of the SAC the complete algorithm will be realised as fixed point. Then all signals will be coded as integer.

3.3.2 Possible Fault Models for the SAC

3.3.2.1 Modelling Faults

This section describes a number of modelling faults that occurred in previous projects at FFA. We discuss how these modelling faults can be simulated by mutations or faults. By applying these mutations to the model, a respective test-vector that detects faults of this kind can be generated.

Error	Description
Ambiguous Definition of Variables	The data type of a variable allows an ambiguous interpretation. For instance, a conditional statement `if(b) then…` exposes the same behaviour for all `b>0.` If the intention of the designer was to encode more information in b than just a Boolean decision, then this information is lost. **Corresponding Mutation:** This can be covered by a modification of the type or by a mutation of the conditional expression (see 4.1.3□o).
Overflow of an instable control loop	The output of a control loop exceeds the range representable by the datatype used to represent the result. This error may occur only after a long period of time (several minutes, exceeding the typical runtime of a test-case). **Corresponding Mutation/fault:** This error can only be caught by generating long test-cases. A mutation that adds behaviour triggered by an overflow can be added in order to force the generation of such a test-case.

[1] The sign information comes with additional signal.

Missing constraints for the allowed range of a value	A parameter is intended to range over a certain set of values. If this range is not enforced by a guard, invalid values may be provided. **Corresponding mutation/fault:** This problem can be detected by modifying the range of input values (see 4.1.3).
Wrong signal attached	A wrong signal is attached to a switch or a Simulink block. **Corresponding mutation/fault:** This can be covered by a variable or signal permutation (see 5.1.1).
Time is used as a parameter	A timer, starting from 0 and counting up, is used as a parameter. This may eventually result in an overflow. **Corresponding mutation/fault:** This problem can be detected by mutating the input signal, e.g., starting with a higher counting value such that the overflow occurs earlier (see 4.2.2.2).

3.3.2.2 Programming Faults

Error	Description
Wrong parenthesis in an expression	The parenthesis of an expression computing an offset were wrongly placed. The offset corresponded to the aging of a component, and therefore the error was only detected after a long time. **Corresponding mutation/fault:** If the specification contains the aging parameters of the component, then a respective test case can be generated by mutating the operators of the expression.

FFA implements their software mainly using Simulink models. Parts of these models are implemented in a traditional programming language, such that the mutations discussed in 5.1.1 are applicable.

3.4 Agricultural Domain: Fault Model for ISOBUS Implementation

Possible faults in the ISOBUS standard implementation and agricultural systems modelling are mainly related to software syntax errors or to wrong interpretation and modelling of state machines. As the ISOBUS protocol can be implemented both by using state charts or by manually writing code typical faults are the ones already mentioned in Section 3.3.2 with respect to the Automotive Demonstrator.

3.4.1.1 ISOBUS Faults

This section describes a number of ISOBUS functioning faults that may occur due to wrong protocol implementation or programming fault.

Error	Description
Timer Timeout	ISOBUS protocol states that every node connected to the network must send particular maintenance messages within a certain slot of time. If an ECU does not send its maintenance message and does not respect timeouts, it is disconnected from the network.
Bus Traffic Overflow	If too many messages are sent on the network, many ECUs may not work properly. Typically if bus traffic is over 60% some messages are lost during communication.
CAN Buffer Size	If CAN Buffer is not well software or hardware implemented and the message queue is too short, some messages may be lost introducing communication errors.
Messages Priority	If too many ECUs send their messages without waiting for Virtual Terminal replies, Virtual Terminal stops working properly and the network cannot be certified.

4 Taxonomy of Fault Models

This section provides a classification of Software Mutations and Failure Modes. While software mutations are syntactic alterations of the source code and have no domain specific characteristics, injected faults and errors are typically tied to the specific components and peculiarities of the domain (e.g., heavy ion injection is a technique typically applied in the avionic or space domain). A fault is defined as a time-location pair, meaning that two faults injected at the same location, but at two different instants in time, could have a completely different effect on the system. Thus, the time aspects are important and faults are thus injected during execution of the target system. A fault model (also denoted as a failure mode in e.g. safety standards) can be defined by means of the number of faults, the time aspect and the fault type. An example of a fault model is a *single transient bit-flip fault*. While the granularity of software mutations is usually restricted to single expressions, operators, or code lines, faults can occur at the electrical circuit level, the gate level, functional levels, component level or the sub-component level, resulting in a large variety of fault injection techniques aimed at different levels of abstraction.

Primarily, we group our fault models into mutations and failure modes (simulated by fault injection). This classification is motivated by the different nature of these faults discussed in the Introduction. A classification applying to both mutations and injected faults is the division of failure effects into transient, intermittent and permanent faults.

4.1 Taxonomy of Mutations

Software mutations are simple syntactical alterations of the source code. Though the mutation operators may be inspired by typical mistakes made by programmers, the modifications are typically local (i.e., the methodology does not address realistic programming mistakes, which are usually more complex). The incentive of introducing mutations is to help the generation of test cases that achieve a certain structural coverage of the code.

Models can be used either to only (semi-)formally express the requirements or to also generate the application code from them. Tests derived from the former use are black box tests, while the latter case results in white box tests. Model mutation for white box testing, as for software mutation, should achieve a structural coverage of the code.

Model mutation for black box testing, as in the UML/action systems track of MOGENTES, should achieve a good coverage of the modelled behaviour – the test case generation should find good equivalence classes for the requirements expressed in the model. The means to do this are, as well as when mutating source code, small, local changes to the model – again based on the coupling effect and the competent programmer (and modeller) hypothesis, where the latter also allows for assuming that a correct implementation of the model is more or less a refinement of the model.

In addition to mistakes made by the programmer, there is also the possibility of a misunderstanding regarding the meaning of the model between the modeller and the programmer. For UML this is especially true because the UML 2 standard gives a set of so called semantic variation points, where the semantic of UML models is intentionally underspecified to give leeway for domain specific profiles. In MOGENTES, we chose a certain semantic interpretation for the variation points relevant for the demonstrators (see D2.2b [BME, 2009]).

Furthermore, there are some concepts in UML where the specified semantic differs from the interpretation taken on the first impression, For example, it is sometimes missed that a change trigger only fires on a change – if the condition of the change trigger of a leaving transition of a state is already true when entering the state, the change trigger never fires, although this could be the behaviour intended by the modeller or read by the programmer.

In the UML/action systems track, these two groups of mistakes can be covered by not introducing mutations to the model itself, but to the transformation process from the model to the action system.

4.1.1 Modification Categories

We first describe the fundamental categories of mutations or modifications that can be applied to a development artefact (like a program or a model). For each of the following mutations, it has to be taken into account that the modification must not destroy the well-formedness of the artefact at hand: the artefact must still be syntactically correct and type-correct (otherwise, the modified development artefact would not be useful for further processing like generation of test cases, etc.). This means that the mutation categories have to be instantiated carefully for individual programming languages and modelling formalisms.

The first two kinds of modifications represent the most basic mutations:

- *Insertion:* Add a new syntactic element to a program or model, e.g., insert further statements, expressions, or transitions.
- *Deletion:* Similarly, it is possible to remove a part from a program or a model.

The following two categories combine multiple insertion or deletion modifications:

- *Replacement:* Replace a syntactic element of a development artefact with a different element, e.g., change the operators of an expression.
- *Exchange:* Swap two or multiple syntactic elements of a development artefact, e.g., change the order of statements.

4.1.2 Granularity of Mutations

An orthogonal dimension is given by the level on which mutations are applied. Possible levels are determined by the kind of the development artefact and include the following (starting with the finest granularity and ending with the coarser ones):

- Mutation of individual variable occurrences, symbols, operators, literals, etc.
- Mutation of expressions, sub-expressions, triggers of transitions, etc.
- Mutation of statements (which can be either atomic or compound).
- Mutation of transitions, states, etc.

It can be observed that coarser mutations are often more difficult to apply than finer mutations, because fewer restrictions exist on how an element of a development artefact are modified. If it is allowed to replace statements of a program by arbitrary other statements, for instance, the number of possible modifications will be enormous and quickly grow unmanageable.

4.1.3 Examples: Mutation of Imperative Programs

We give a few examples of mutation that can be applied to programs in common programming languages. A more detailed account is given in Section 5, where also mutation of graphical models is discussed.

- *Statement Deletion (or Insertion):* The behaviour of the program is changed by deleting (or inserting) an additional statement (or gate/block) into the code. Statement coverage (for the affected instructions or code blocks) is a necessary (but not sufficient) requirement to detect this change.
- *Modification of Boolean Sub-Expressions in Conditions:* This refers to changing (or fixing the value) of Boolean sub-expressions that occur in the program. Condition coverage for the modified expressions is a necessary (but not sufficient) requirement for the test suite to detect this change.
- *Modification of Operators:* All mutations that change the operator of an expression (possibly a Boolean or arithmetic expression) while leaving the operands unchanged. As a special case, however, also permutation of the operands in an expression can be seen as a change of the operator (e.g., replacement of < by >).
    - *Modification of Relational Operators:* The outcome of conditions is changed by replacing Boolean relations with other ones, e.g., < is replaced by >= or <=. This may be particularly useful to guarantee that a sufficiently thorough boundary value analysis is performed.
    - *Modifications of Arithmetic Operators:* Arithmetic operations are replaced by other ones, e.g., an addition in a term may be replaced by a subtraction.
    - *Modifications of Language Specific Operators:* The concept of changing operators is extremely general. For instance, in ANSI-C it is possible to change the type-conversion operators. (An incorrect type-conversion was the cause of the Ariane 5 catastrophe.)
- *Modification of Data Types or Data Value Ranges:* Data types or allowed value ranges variables are changed – usually to a more restrictive one.

4.2 Failure Behaviour

Traditionally, fault injection is used to simulate hardware faults. The term *hardware fault* refers to a physical fault (while mutations simulate algorithmic faults). We emphasise that hardware faults can be simulated by using fault injection mechanisms implemented in the model (MIFI) by using failure mode functions (FMFs) presented in Section 5.2, implemented by extra software (SWIFI), or implemented by using extra hardware (HIFI). Furthermore, hardware faults can be simulated at different levels of abstraction. A distortion of a signal may be caused by a bit-flip fault at the gate-level (i.e., in silicon), on by electromagnetic interference disturbing a transmission channel, or may even be caused by a defective beacon or track switch. Despite the traditional focus on hardware, the failure modes discussed in this deliverable are not restricted to hardware faults. We also model failures at a higher level of abstraction, such as communication failures. Different failure behaviour presented in the following sub sections are implemented in the proposed FMFs in Section 5.2.

4.2.1 Categorisation of Fault Persistence

- *Transient Faults:* faults that may be caused by events such as overheating, a glitch in the power or transmission line, or subtle software defects. Even if the failure cause is permanently present (such as a software bug), the effect caused by that mechanism may actually be transient (e.g., the software may return wrong results for specific inputs, but may work according to specification for others).
- *Intermittent Faults:* malfunctions that occur periodically (i.e., from time to time). The intervals of occurrence may be regular or irregular.
- *Permanent Faults:* Permanent faults may occur if a component of the system is permanently damaged or non-functional. This may for instance result from the failure of an entire component (e.g., a failing engine), from less severe causes such as a fused connector or pin, or from software consistently misbehaving for the whole domain of input values. Note, that mutations (as described in Section 4.1) can be considered as permanent faults on the level of software.

4.2.2 Fault Categories

In the following sections, we outline in which ways faults can affect the behaviour of a program or system.

4.2.2.1 Suppressed Behaviour

- *Lost Signals or Events:* Lost signals or events (e.g., caused by a failed transmission) may result in failure to execute certain routines of the system. This case is different from the Fail Silent case, because the signal is not intentionally suppressed.
- *Stuck Signals:* In this case, a signal is restricted to a fixed value. A fault of this kind can be introduced by a failing gate.

4.2.2.2 Modified Behaviour

- *Modified Signals or Output:* This refers to signals or outputs that differ from the expected/specified behaviour. A typical example of this may be a flipped bit in a memory cell or CPU register which manifests as a value failure on the output, a perturbation of a transmission, or a component acting as "babbling idiot".
- *Changing the Specified Range of Signals or Output:* By changing the range of a signal or output of a component we can add additional behaviour to a system. This may happen if component reports values outside of its specified limits, e.g., a thermal sensor reporting an unrealistically high (or low) temperature.
- *Delayed or Premature Actions or Events:* This describes the situation in which an action is not executed according to the predetermined schedule (e.g., delayed signals). Possible causes for delayed behaviour are external perturbation, disruption (e.g., a temporary power outage), or stress (e.g., a large amount of data to process). Premature actions can be caused by clock skews.

4.2.2.3 Additional Behaviour

A fault in this class introduces additional behaviour into a system. This may be falsely triggered events (e.g., caused by an external perturbation of a transmission) or even non-deterministic behaviour of a component.

4.2.3 Abstraction Levels of Fault Models

Similarly to mutations (Section 4.1.2), faults can be modelled on different levels of abstraction:

- *Bit-level:* Faults that cause individual bits of data, memory, registers, etc. to be changed. This is the lowest level of abstraction and models, e.g., faults due to electro-magnetic inference.
- *Variable or signal level:* Faults that affect the value of variables (representing e.g., Booleans, integers, or reals by means of floating-point numbers) or signals of a program or system. On this level of abstraction, faults like the erroneous amplification of values can be modelled that would be difficult to capture on the bit-level.
- *Higher level, e.g., the communication level:* On an even higher level of abstraction, faults like message loss of communication channels can be modelled.

5 Fault Model Library

5.1 Mutations

5.1.1 Mutation of Imperative Textual Programs

The classical set of mutations for imperative languages is given in [King et al, 1991] and comprises 22 mutations designed for the Fortran programming language. Most of the mutations are also relevant for low-level languages like C and are given in the table below (adapted to C).

Mutating type or operation names usually results in compiler or linker errors, and is therefore not appropriate for generating executable mutants. Modifying keywords like replacing "private" by "public" in Java or C++ is a bit different, because some of these mutations will result in executable code, but will only potential effect in combination with other mutations. For instance, the given example has only effect if somewhere an object accesses a foreign class variable which is sheltered with "private" in the original software. Since mutations in general address single mutations, keyword mutation is not further considered.

Note that these considerations also apply for OCL. The listed fault model library also applies with the modification that "variable" is to be replaced by "model element".

Mutated element	**Mutation name**	**Mutation description**	**Remarks**
Arithmetic operator	Arithmetic operator replacement	Each occurrence of one of the operators +, –, *, /, % is replaced by each of the other operators. In addition, each is replaced by the operators LEFTOP and RIGHTOP. LEFTOP returns the left operand (the right is ignored), RIGHTOP returns the right operand.	An operator is never replaced with itself.
Array reference	Constant for array reference replacement	Each array reference in a program unit (or function) is replaced by each constant visible in the unit, provided that the base type of the array and the type of the constant are both arithmetic.	Left-hand sides of assignments are not replaced.
Array reference	Comparable array name replacement	In each array reference, the array name is replaced by the name of every other array of compatible type.	An array is never replaced with itself.
Constant	Constant replacement	Each constant value is modified slightly to emulate domain perturbation testing. The change to the value depends on the constant’s type: • Each integer constant is both incremented by 1 and decremented by 1. • Each non-zero real and double precision constant is incremented and decremented by 10 per cent of its value; zero is replaced by 0·01 and –0·01. • Each Boolean constant is replaced by its complement.	
Variable occurrence	Constant for	Each scalar variable in a program unit	Left-hand sides of

	scalar variable replacement	is replaced by each constant visible in the same unit, provided that the types of the variable and the constant are both arithmetic.	assignments are not replaced.
Goto statement	Goto label replacement	The label of a goto statement is replaced by every label in the same function.	A label is never replace by itself.
Logic operator	Logical operator replacement	Each occurrence of one of the boolean operators &&, ||, ==, != is replaced by each of the other operators. In addition, each is replaced by the operators LEFTOP and RIGHTOP.	An operator is never replaced with itself.
Relational operator	Relational operator replacement	Each occurrence of one of the relational operators <, >, <=, >=, ==, != is replaced by each of the other operators. In addition, each is replaced by the operators LEFTOP and RIGHTOP.	An operator is never replaced with itself.
Statement	Return statement replacement	Each statement in a function is replaced by a return statement.	
Statement	Statement analysis	Each statement at the beginning of a basic block is replaced by assert(false).	
Array reference	Scalar variable for array reference	Each array reference in a program unit is replaced by every scalar variable of compatible type appearing in the program unit.	A variable is not used as a replacement on the right side of an assignment statement when so doing would cause the two sides to become identical.
Constant	Scalar for constant replacement	Each constant in a program unit is replaced by every scalar variable of compatible type visible in that unit.	A variable is not used as a replacement on the right side of an assignment statement when so doing would cause the two sides to become identical.
Statement	Statement deletion	Each statement is deleted.	This does not apply to variable declarations, deletion of which would produce ill-formed programs.
Constant	Source constant replacement	Each arithmetic constant in a program unit is replaced by every arithmetic constant visible in that unit.	A constant is never replaced with itself.
Variable	Scalar variable replacement	Each scalar variable in a program unit is replaced by each scalar variable of compatible type visible in the same unit.	
Expression	Unary operator insertion	Each arithmetic expression is negated, incremented by 1 and decremented by 1. Each logical expression is complemented.	

5.1.2 UML Model Mutation

Similar to the situation with syntax violations for programming languages, a model mutation must not lead to conflicts with the meta-model used. Furthermore, good mutation operators result in test cases with a good coverage in relation to the number of mutated systems resulting from applying the operator.

The mutation operations described below are preselected based on the results of [Black, 2000], as far as a conclusion by analogy is applicable to mutation of the UML model elements and based on considerations reflected in the remarks column of the tables. Also, mutations which are not of interest for the MOGENTES demonstrators may be noted, but are not described in the tables.

5.1.2.1 Class Mutation

Mutating the structural features of the model usually makes it difficult to leave the model syntactically correct or, when adapting references to the changed elements, leads to too substantial changes in the behaviour. If this is the case, one major benefit of mutations – a fine-grained coverage – is lost. Therefore, only few mutation operators, inspired by common programming errors, are listed here.

Mutated element	**Mutation name**	**Mutation description**	**Remarks**
Literal Values (are a special case, since they are used in many other model elements, e.g. constants, initial values for variables, OCL or AGSL expressions and are not limited to classes.)	Literal Value Variation Step	Add/substract one precision unit to/from the literal	For Integers, the precision unit is one. For real numbers, the precision unit can be either a step of the least significant digit in the code or of the least significant bit of the used data type. Alternatively, this could be set application/domain specific, but this would need a manual intervention when mutating.
	Literal Value Variation Factor	Increase/decrease the literal value by one order of magnitude (multiply with/ divide by 10)	Reflects a wrong positioned decimal point.
Constant	Constant Variation Step	Add/substract one precision unit to/from the constant	See same operator on literal values
	Constant Variation Factor	Increase/decrease the constant by one order of magnitude (multiply with/ divide by 10)	See same operator on literal values
	Set Constant to Zero	Reduce Constant value to Zero	Reflects an unset numerical constant.
Property	Minimally Reduce Data Type Range	Reduce the data type to the next smaller compatible data type.	
	Maximally Reduce Data Type Range	Reduce the data type to the smallest compatible data type.	
Method	Apply Possible Body Mutations	Modify the body of the method by applying one of the mutations described in 0 (OCL Mutation) and 5.1.2.4 (AGSL Mutation)	

5.1.2.2 State Machine Mutation

5.1.2.2.1 State Mutations

State deletion and insertion or even state splitting would be too large a modification, and can be replaced by mutating the incoming transitions (see also 2.1). Mutation therefore is limited to entry and exit action behaviour.

Mutated element	Mutation name	Mutation description	Remarks
State Entry/Exit Action	Remove State Entry/Exit Action		
	Exchange State Entry/Exit Action Method	Replace the entry/exit action method call with the method called in another entry/exit action.	
	Apply Possible Body/Method Call Mutations	Modify the body/method call of the entry/exit action by applying one of the mutations described in 0 (OCL Mutation) and 5.1.2.4 (AGSL Mutation)	

5.1.2.2.2 Transition Mutations

Mutated element	Mutation name	Mutation description	Remarks
Transition	Remove Transition	Remove the transition completely.	May prove too coarse grained. Is equivalent to setting the guard of the transition to false. Any test case using this transition will potentially detect that mutation. Corresponds to testing normal behaviour.
Transition Source/Target	Exchange Transition Source/Target	Move the source/target of the transition to another state in the state machine (including the target/source itself).	The introduced loops may lead too syntactically identical mutations. For initial and final states, only the distant end of the transition is moved.

5.1.2.2.3 TransitionTrigger Mutations

Mutated element	Mutation name	Mutation description	Remarks
Transition Trigger	Exchange Transition Trigger	Replace the trigger of the transition with the trigger of an arbitrary other transition in the model (including removing the trigger and stereotyping the transition as *triggerless*)	The triggerless stereotype will be described in D3.2b due month 30)
Signal Trigger	Exchange Trigger Signal	Replace the signal of the trigger with an arbitrary signal where the class has a receiption defined.	This may cause model errors when properties of the signals are used in the transition effect.
Time Trigger	Minimally Change Time Trigger Duration	Increase/Decrease time trigger duration by one unit	
	Substantially Change Time	Increase/Decrease time trigger duration by one order of magnitude	

	Trigger Duration	(factor 10).	
Change Trigger	Modify Change Trigger Expression	Modify the expression of the change trigger by applying one of the mutations described in 0 (OCL Mutation).	
Call Trigger	Exchange Call Trigger Method	Replace the method of the trigger with an arbitrary method of the class.	This may cause model errors when parameters of the methods are used in the transition effect.

5.1.2.2.4 Transition Guard Mutations

Mutated element	Mutation name	Mutation description	Remarks
Transition Guard	Remove Guard	Remove the guard from the transition.	Is equivalent to setting the guard to true. Any test data which evaluates the original guard to false will potentially detect that mutation. Corresponds to testing absence of unwanted behaviour.
	Invert Guard	Logically negate the guard expression	Any test case causing the respective trigger in the source state will potentially detect that mutation.
	Modify Guard Expression	Modify the guard expression by applying one of the mutations described in 0 (OCL Mutation).	

5.1.2.2.5 Transition Effect Mutations

Mutated element	Mutation name	Mutation description	Remarks
Transition Effect	Remove Effect	Remove the effect from the transition	
	Exchange Effect Method	Replace the transition effect with an arbitrary effect of another transition	
	Modify Effect Body	Modify the body/method call of the effect by applying one of the mutations described in 5.1.2.4 (AGSL Mutation)	

5.1.2.3 OCL Mutation

Use of expressions from a subset of OCL is supported in transition guards, change expressions and post conditions. The mutations are similar to the code mutations in Section 5.1.1:

Mutated element	Mutation name	Mutation description	Remarks
Expression	Fix Expression	Replace the whole expression with true or false	
Expression	Negate	Negate the whole expression	

	Expression		
Logical Subexpression	Fix Subexpression	Replace an arbitrary subexpression with true or false	
Logical Subexpression	Negate Subexpression	Negate an arbitrary subexpression	
Boolean Operator	Modify Boolean Operator	Change a binary Boolean operator to another.	
Relational Operator	Modify Relational Operator	Change a relational operator to another.	
Arithmetic Operator	Modify Arithmetic Operator	Change an arithmetic operator to another.	
Set Operator	Modify Set Operator	Change a set operator to another (union, intersection, minus)	
Quantifier Operator	Modify Quantifier	Change a quantifier operator to another (exists, forall, one).	
Operand	Replace Operand	Replace an operand, by another syntactically legal operand.	This will be possibly left out since, in contrast to the notation used by [black], in OCL it can get very difficult to replace an operand (except a constant/literal value, see further above) with a sensible set of syntactically legal other operands.

5.1.2.4 AGSL Mutation

Activity and Guard Specification Language (AGSL) is a small language which in the context of MOGENTES is used to define activities in UML state machine diagrams. The used subset is limited to:

- Calls to methods of associated objects with parameters taken from object attributes or, in the context of signal triggers, received signal properties.
- Sending signals to associated objects with signal properties taken from object attributes or, in the context of signal triggers, received signal properties.
- Assigning values built from object attributes or, in the context of signal triggers, received signal properties to object attributes

Basically, the general code mutation operations apply.

Mutated element	**Mutation name**	**Mutation description**	**Remarks**
Arbitrary statement	Remove Statement	Remove a statement in a sequence of statements.	
Arbitrary statement	Reorder Statement	Move a statement to another position in the sequence.	
Send Signal/Call Method	Fix Parameter/Property	Replace a method call parameter or property of a sent signal with a literal value.	
Send Signal/Call	Modify	Replace a method call parameter or property of a sent signal with another	

Method	Parameter/Property	of the same type.	
Assignment Statement	Modify Operator	Exchange an operator with a compatible one.	
Assignment Statement	Fix Operand	Replace an operand with a literal value.	
Assignment Statement	Modify Operand	Replace an operand with a variable, call parameter or signal property of the same type.	
Assignment Statement	Fix Result	Replace the whole right side expression with the default value of the left hand side.	

5.1.3 UML Semantic Interpretation Mutation

This section lists a set of semantic variations of state machines in the UML specification, the chosen interpretation within MOGENTES and if there are interpretations which will be used as mutations. Furthermore, treatment of one frequent misunderstanding of UML state machine semantics and of one MOGENTES specific construct are described, together with a possible mutations.

In contrast to the mutations described above, mutations reflecting a different semantic interpretation affect the whole system and therefore there is usually only one single mutant to consider per mutation operator, instead of one mutant for each model element possibly affected by a mutation.

5.1.3.1 Event Reordering

Standard:

The ordering of the events in the input pool is a semantic variation.

According to the standard, objects process event occurrences from their event pool (signal events, timer events, call events, change events) in arbitrary order.

MOGENTES:

Within MOGENTES, FIFO processing is assumed for the event pool – which is a less non-deterministic view on the system and usually leads to more stable systems, but could also lead to more deadlocks.

Mutation(s):

With the support of non-determinism in action systems, it is possible to produce mutated action systems conforming to a randomly choosing event pool behaviour.

5.1.3.2 Event Preservation

Standard:

If an event in the pool satisfies no triggers at a wait point, it is a semantic variation point what to do with it.

It is a semantic variation whether an event is discarded if there is no appropriate trigger defined for them.

MOGENTES:

Unused events could be stored indefinitely in the pool – for embedded systems, this could easily lead to overflows. Within MOGENTES, events that are not consumed by a trigger, are dropped. Additionally, explicitly deferring triggers is not supported.

Mutation(s):

A mutation where unused events are left in the pool can be implemented together with event reordering described above. This mutation may be of little use in the context of test case generation, since it easily leads to overflows or very long traces.

5.1.3.3 Event Processing

Standard:

It is a semantic variation whether one or more behaviours are triggered when an event satisfies multiple outstanding triggers.

MOGENTES:

Within MOGENTES, in each orthogonal region one trigger per event occurrence is used. If within an orthogonal region, after taking priorities into consideration, still more than one trigger is satisfied, one is non-deterministically chosen.

Mutation(s):

Choose all available triggers for the event (after taking priority due to state to sub-state relations into consideration).

5.1.3.4 Signal Transmission Delay, Signal Loss

Standard:

The means by which requests are transported to their target depend on the type of requesting action, the target, the properties of the communication medium, and numerous other factors. In some cases, this is instantaneous and completely reliable while in others it may involve transmission ***delays of variable duration****,* ***loss of requests****, reordering,* ***or duplication****.*

MOGENTES:

Since none of the MOGENTES demonstrators are distributed in a form where signal delay or unreliable communication can occur legally, signal transmission is done reliably and immediately.

Mutation(s):

None.

5.1.3.5 Signal Reordering

Standard:

The means by which requests are transported to their target depend on the type of requesting action, the target, the properties of the communication medium, and numerous other factors. In some cases, this is instantaneous and completely reliable while in others it may involve transmission delays of variable duration, loss of requests, ***reordering****, or duplication.*

MOGENTES:

Signals are delivered in the order they are sent in.

Mutation(s):

No additional mutations; this is covered by the event reordering mutation since its possible effects are reduced anyway when there is no transmission delay.

5.1.3.6 Composition Aggregation

Standard:

There are several semantic variation points dealing with composition/aggregation.

MOGENTES:

These semantic variation points are not relevant within MOGENTES since dynamic object creation and deletion is not used (i.e. limited to creation at initialisation) – which is also true for the majority of embedded systems in general.

Mutation(s):

None

5.1.3.7 Change Trigger Occurrence

Standard:

It is a semantic variation when the change expression is evaluated. For example, the change expression may be continuously evaluated until it becomes true.

MOGENTES:

Change expressions are continuously evaluated until they change from false to true.

Mutation(s):

None.

5.1.3.8 Change Trigger Removal

Standard:

It is further a semantic variation whether a change event remains until it is consumed, even if the change expression changes to false after a change event.

MOGENTES:

Change events stay in the event queue even when the expression changes to false before the event is consumed.

Mutation(s):

Add the change trigger expression additionally to the guard of the transition, thereby removing the event from the queue without actually taking the transition.

5.1.3.9 Event Consumption Timing

Standard:

No assumptions are made about the time intervals between event occurrence, event dispatching, and consumption. This leaves open the possibility of different semantic variations such as zero-time semantics.

MOGENTES:

Zero-Time semantics is assumed.

Mutation(s):

Before entering an event into the queue, all running timers could be reduced an arbitrary amount of time to emulate delays between occurrence and dispatching.

5.1.3.10 Triggerless transitions

MOGENTES:

For ease of modelling, a transition stereotype *triggerless* is introduced. Transitions stereotyped as triggerless are always immediately taken if its guard is true.

Mutation(s):

a. Change to a transition with a change trigger having the guard expression as change expression
b. Change to a transition without the stereotype, which is taken only if the source state reaches its internal end state.

5.1.3.11 Change Trigger Semantics

Standard:

A change trigger only occurs when the expression changes from true to false.

Misunderstanding:

If a state is entered when the expression of a leaving change triggered transition is already true, it might be expected that the transition is taken immediately, but it is not taken before the expression changes to false and back to true.

Mutation:

Interpret the change triggered transition as *triggerless* as described above, with the guard extended by the change trigger expression.

5.1.3.12 Default Transition Semantics

Standard:

A transition without a trigger is taken when the source state reaches its inner end state or immediately after taking the entry action if there are no sub states.

Misunderstanding:

It could be expected that if the guard is not true when the end state is reached, the transition is taken when the guard condition becomes true,

Mutation:

Treat the transition as a *triggerless* transition (as described above) with the source state changed to the end state of the sub state machine if there are sub states,

5.1.4 Mutations for Action Systems

Because automated test case generation in MOGENTES (partly) is based on action systems, mutations to action systems are of particular interest. In principle action systems comprise two parts: one part defines named actions (roughly speaking named actions are state transformers) that are used in the second part. The idea is that by combining named actions in different ways the system behaviour is defined. Currently, named actions may be combined by sequential, non-deterministic, and prioritized composition in any order. Based on these facts it becomes clear that there are two main ways of how action systems can be mutated. The first alternative is to mutate the composition of named actions, while the second alternative is to mutate the named actions themselves.

Since named actions also serve as atomic entities of observable behaviour of the system under test (SUT), mutating the composition of named actions relates to mutating the “protocol”, i.e. the order in which certain observations are expected from the SUT and controllables can be provided. Therefore these kinds of mutations are similar to the mutation of process algebras and we may re-use some of the mutations introduced in the context of process algebras, e.g. LOTOS, or CSP. For example, ESO (event swap operation) translates into swapping of named actions, EDO (event drop operation) would result in dropping a named action from the composition, and SOR (sequential operator replacement) means replacing the sequential composition operator. More mutation operators for LOTOS may be found in [AichernigDelgado2006] and [WeiglhoferAichernigWotawa2009].

Notice that when mutating the implementation of named actions some of the process-algebrae inspired mutation operators may also be used. As an example, USO (unobservable sequence operator) that makes events of the specification unobservable could be implemented by removing any observable or controllable tag from a named action, making the action internal. In general we are free to apply a larger variety of mutation operators to the implementation of a named action than to the combination of named actions. Because a named action always comprises one guarded command, any mutation operator that changes expressions can be applied. In addition, as actions consist of statements, mutation operators working on statements can be used. Summing up we can say that mutating the implementations of named actions is similar to the mutation of programs.

Within MOGENTES we also find the notion of object-oriented action systems. That said, the supported notion of object-orientation is very limited, e.g., it is lacking any form of late-binding, therefore only few of the commonly used mutation operators for object-oriented programs (cf. [MaOffuttKwon2005]) are useful in this context.

5.2 Failure Mode Functions

A *failure mode function (FMF)* can be seen as a function (or model block) implementing a failure mode by manipulating signals between blocks in a model, or that manipulating operators, to simulate the effect of faults/errors that will lead to a failure (e.g. value failure or a component failure). A general remark valid for all FMFs presented in this section is that at each time step when a failure is not activated the *actual value* is equal to the *nominal value* (non-faulty value); else the actual value assumes the value determined by the FMF. The temporal aspect (transient, intermittent, permanent) of the FMF is defined by a Boolean input to the FMF where a true value means that the FMF is activated. The presented FMFs can be inserted into an executable model in order to try to violate e.g. safety requirements. The work of designing FMFs will be carried out in Task 4.4: Model-Implemented Fault Injection Mechanisms.

5.2.1 Boolean Failure Mode Functions

Name	**Description**	**Pseudo code**

Invert	When the failure occurs the actual value is inverted from the nominal value, i.e. changed from true to false or from false to true.	FMF_Bo_Invert()
		if fi_trigger[n]==TRUE

		output[n] = !input[n]
		else
		  output[n] = input[n]

Off	Forces the actual value to be “false” when the failure occurs.	FMF_Bo_Off()
		if fi_trigger[n]==TRUE
		  output[n] = FALSE
		else
		  output[n] = input[n]

On	Forces the actual value to be “true” when the failure occurs.	FMF_Bo_On()
		if fi_trigger[n]==TRUE
		  output[n] = TRUE
		else
		  output[n] = input[n]

Stuck	When the failure occurs the actual value will hold the nominal value, which the variable had at the time step of failure occurrence and for as long as the failure remains true.	FMF_Bo_Stuck()
		if fi_trigger[n]==TRUE && fi_trigger[n-1]==TRUE
		  output[n] = output[n-1]
		else
		  output[n] = input[n]

Stuck_on_if_on	When the failure occurs at the same time step as the nominal value is on (true) the actual value will become on (true) and remains on as long as the failure remains true. As soon as the failure becomes false the actual value will be equal to the nominal value.	FMF_Bo_Stuck_on_if_on()
		if fi_trigger[n]==TRUE && fi_trigger[n-1]==TRUE
		  if output[n-1]==TRUE
		    output[n] = TRUE
		  else
		    output[n] = input[n]
		else
		  output[n] = input[n]

Stuck_on_if_off	Analogue to Stuck_on_if_on.	FMF_Bo_Stuck_on_if_off()
		if fi_trigger[n]==TRUE && fi_trigger[n-1]==TRUE
		  if output[n-1]==FALSE
		    output[n] = TRUE
		  else
		    output[n] = input[n]
		else
		  output[n] = input[n]

Stuck_off_if_on	Analogue to Stuck_on_if_on.	FMF_Bo_Stuck_off_if_on()
		if fi_trigger[n]==TRUE && fi_trigger[n-1]==TRUE
		  if output[n-1]==TRUE
		    output[n] = FALSE
		  else
		    output[n] = input[n]
		else
		  output[n] = input[n]

Stuck_off_if_off	Analogue to Stuck_on_if_on.	FMF_Bo_Stuck_off_if_off()
		if fi_trigger[n]==TRUE && fi_trigger[n-1]==TRUE
		  if output[n-1]==FALSE
		    output[n] = FALSE
		  else
		    output[n] = input[n]
		else
		  output[n] = input[n]

Delay_hold	The sequence of nominal values will be delayed a number of cycles, which is specified by the user. During the delay time the actual value will be set to the nominal value at time step before failure occurrence.	FMF_Bo_Delay_hold(value)
		slwbuf[value]
		if fi_trigger[n]==TRUE && fi_trigger[n-1]==FALSE
		  slwbuf[1:value] = input[n-1]
		  output[n] = input[n-1]

		if fi_trigger[n]==TRUE && fi_trigger[n-1]==TRUE
		  slwbuf[1:value] << 1
		  slwbuf[value] = input[n-1]
		  output[n] = slwbuf[1]
		else
		  output[n] = input[n]

Delay_repeat	The sequence of nominal values will be delayed a number of cycles, which is specified by the user. At failure occurrence the actual value sequence repeats the nominal value sequence with the specified delay.	FMF_Bo_Delay_repeat(value)
		if fi_trigger[n]==TRUE
		  output[n] = input[n-value]
		else
		  output[n] = input[n]

Short_circuit	Makes two variables – A and B – dependent of each other. When the failure occurs, the actual values for A and B will become “true” if at least one of them has a nominal value being “true”. In all other cases, their actual values will remain the same as their nominal values.	FMF_Bo_Short_circuit()
		if fi_trigger[n]==TRUE
		  output1[n] = input1[n] || input2[n]
		  output2[n] = input1[n] || input2[n]
		else
		  output1[n] = input1[n]
		  output2[n] = input2[n]

Open_circuit	This FMF corresponds to an unconnected wire, i.e., the signal can take an arbitrary, non-deterministic value.	FMF_Bo_Open_circuit()
		if fi_trigger[n]==TRUE
		  temp = rand(0,1)
		  if temp < 0.5
		    output[n] = FALSE
		  else
		    output[n] = TRUE
		else
		  output[n] = input[n]

Swap_value	This failure mode (when enabled) will swap the values for two variables. For variables A and B the failure effect is: temp = A; A = B; B = temp.	FMF_Bo_Swap_value()
		if fi_trigger[n]= TRUE
		  output1[n] = input2[n]
		  output2[n] = input1[n]
		else
		  output1[n] = input1[n]
		  output2[n] = input2[n]

Random	This failure mode (when enabled) will randomly set the actual value to true or false.	FMF_Bo_Random()
		if fi_trigger[n]==TRUE
		  temp = rand(0,1)
		  if temp < 0.5
		    output[n] = FALSE
		  else
		    output[n] = TRUE
		else
		  output[n] = input[n]

Table 5-1: Boolean fault modes

5.2.2 Integer/Real Failure Mode Functions

Name	Description	Pseudo code

Constant	Forces the variable’s actual value to a constant value, provided as a parameter, when the failure occurs.	FMF_IR_Constant(value)
		if fi_trigger[n]==TRUE
		  output[n] = value
		else

		output[n] = input[n]

Amplification	The actual value is multiplied by a fixed value, provided as a parameter, at each time step. Whenever the failure switches to “not active” the actual value is reset to the nominal value.	FMF_IR_Amplification(value)
		if fi_trigger[n]==TRUE
		  output[n] = input[n]*value
		else
		  output[n] = input[n]

Amplification_range	Amplify the variable’s nominal value by a randomly selected value when the failure occurs. A new random value – chosen from an interval, [min, max], where min and max is specified by the user – is decided at each time step.	FMF_IR_Amplification_range(value1,value2)
		if fi_trigger[n]==TRUE
		  value=rand(value1,value2)
		  output[n] = input[n]*value
		else
		  output[n] = input[n]

Drift	Makes the actual value drift away from the nominal value by a fixed amount, provided as a parameter, at each time step. Whenever the failure switches to “not active” the actual value is reset to the nominal value.	FMF_IR_Drift(value)
		if fi_trigger[n]==TRUE && fi_trigger[n-1]==FALSE
		  temp = value
		  output[n] = input[n]+temp
		else if fi_trigger[n]==TRUE
		  output[n] = input[n]+temp
		  temp = temp + value
		else
		  output[n] = input[n]

Decrease	Decreases the variable’s nominal value by an amount, provided as an integer parameter, when the failure occurs.	FMF_IR_Decrease(value)
		if fi_trigger[n]==TRUE
		  output[n] = input[n]-value
		else
		  output[n] = input[n]

Decrease_range	Decreases the variable’s nominal value by a randomly selected value when the failure occurs. A new random value – chosen from an interval, [min, max], where min and max is specified by the user – is decided at each time step.	FMF_IR_Decrease_range(value1,value2)
		if fi_trigger[n]==TRUE
		  value=rand(value1,value2)
		  output[n] = input[n]-value
		else
		  output[n] = input[n]

Increase	Increases the variable’s nominal value by an amount, provided as an integer parameter, when the failure occurs.	FMF_IR_Increase(value)
		if fi_trigger[n]==TRUE
		  output[n] = input[n]+value
		else
		  output[n] = input[n]

Increase_range	Increases the variable’s nominal value by a randomly selected value when the failure occurs. A new random value – chosen from an interval, [min, max], where min and max is specified by the user – is decided at each time step.	FMF_IR_Increase_range(value1,value2)
		if fi_trigger[n]==TRUE
		  value=rand(value1,value2)
		  output[n] = input[n]+value
		else
		  output[n] = input[n]

Ramp_down	Makes the actual value decrease by a fixed amount, provided by the user, at each time step. The actual value, whenever the failure is active, will never become lower than a limit value, which is provided by the user.	FMF_IR_Ramp_down(value,limit)
		if fi_trigger[n]==TRUE
		  temp = input[n]-value
		  if temp<limit
		    output[n] = limit
		  else
		    output[n] = temp
		else
		  output[n] = input[n]

Stuck	Makes the actual value to keep the nominal value – at the time step when the failure occurred – for as long as the failure is active. When the failure is de-	FMF_IR_Stuck()
		if fi_trigger[n]==TRUE && fi_trigger[n-1]==TRUE

	activated the actual value is reset to the nominal value.	output[n] = output[n-1] else output[n] = input[n]
Delay_hold	The sequence of nominal values will be delayed a number of cycles, which is specified by the user. During the delay time the actual value will be set to the nominal value at the time step before failure occurrence.	FMF_IR_Delay_hold(value) slwbuf[value] if fi_trigger[n]==TRUE && fi_trigger[n-1]==FALSE slwbuf[1:value] = input[n-1] output[n] = input[n-1] if fi_trigger[n]==TRUE && fi_trigger[n-1]==TRUE slwbuf[1:value] << 1 slwbuf[value] = input[n-1] output[n] = slwbuf[1] else output[n] = input[n]
Delay_repeat	The sequence of nominal values will be delayed a number of cycles, which is specified by the user. At failure occurrence the actual value sequence repeats the nominal value sequence with the specified delay.	FMF_IR_Delay_repeat(value) if fi_trigger[n]==TRUE output[n] = input[n-value] else output[n] = input[n]
Random	When the failure is enabled the actual value will be selected randomly at each time step within an interval [min, max], where the min and max values are specified by the user.	FMF_IR_Random(value1,value2) if fi_trigger[n]==TRUE value = rand(value1,value2) output[n] = value else output[n] = input[n]
Swap_value	This failure mode (when enabled) will switch the values for two variables. For variables A and B the failure effect is: temp = A; A = B; B = temp.	FMF_IR_Swap_value() if fi_trigger[n]= TRUE output1[n] = input2[n] output2[n] = input1[n] else output1[n] = input1[n] output2[n] = input2[n]

Table 5-2: Integer/real fault modes

5.2.3 Failure Mode Functions at Operator Level

Name	**Description**	**Pseudo code**
Change_operator	When the failure is enabled the selected operator will be replaced by another operator. This scenario could happen if e.g. a bit-flip occurs in the code area of the memory or in a CPU register when an instruction is loaded. This scenario could also occur if the designer is using a wrong block (SW bug).	FMF_Op_Change_operator(value) if fi_trigger[n]= TRUE output[n] = op(input1[1],input[2],value) else output[n] = op(input1[1],input[2])

Table 5-3: Operator level fault modes

5.2.4 Failure Mode Functions at Bit Level

Name	**Description**	**Description**
Flip_bits	This FMF will flip (alter the logical value for) one or more bits in the representation of the actual value.	FMF_Bi_Flip_bits(mask) if fi_trigger[n]==TRUE temp = datatype2hex(input[n])

		temp = temp ^ mask
		output[n] = hex2datatype(temp)
		else
		output[n] = input[n]

Set_bits	This FMF will set one or more bits to a specific bit pattern in the representation of the actual value.	FMF_Bi_Set_bits(mask)
		if fi_trigger[n]==TRUE
		output[n] = hex2datatype(mask)
		else
		output[n] = input[n]

Table 5-4: Bit level fault modes

5.2.5 Failure Mode Functions at Communication Level

This section discusses a number of FMFs that model communication *errors*. Possible causes for such errors may be low level hardware faults (which can be simulated by using some of the FMFs discussed in the previous sections), synchronisation faults, software bugs, or high traffic on the network. Communication errors are modelled at a high level of abstraction and some of the errors listed in this section can be simulated by using one or more of the FMFs described above. Simulation of communication errors may be used to verify protocols or the reliability of a communicating system.

5.2.5.1 Violation of Functional Integrity Requirements

The FMFs listed below introduce distortions of the transmitted data that result in a violation of the functional integrity requirements of the system. Therefore, FMFs of this kind are domain and application specific.

Category	Name	Description
Erroneous Information Received	Transmitter Identifier Error	The identifier in a message referring to the transmitting subsystem is altered.
	Data Type Error	The type of the transmitted data is modified, resulting in a violation of the data integrity of the system.
	Data Value Error	The value of the transmitted data is modified, resulting in a violation of the data integrity of the system.
Time errors at receiver	Out-dated Data or Data Not Received in Due Time	The transmission of a message is delayed, such that the data is out-dated upon reception, or the message is not received within the specified time frame.
	Loss of Communication After a Predefined Delay	The communication is interrupted after a predefined delay.

Table 5-5: Violation of functional integrity requirements

5.2.5.2 Repetition of a Message or Part of the Message

A duplicate message is sent. The cause of this error may be the loss of a message that confirms the reception of a package.

5.2.5.3 Quantitative Data integrity errors

Data integrity errors can be caused by malfunctioning transmission hardware and can typically be detected using the CRC error detection mechanism. These faults may occur due to failing non-trusted transmission hardware or may be caused by random errors due to external influence (e.g. EMI) on the transmission media.

Name	Description
Interrupted Transmission Line	The transmission line is interrupted, resulting in a loss of messages or a complete termination of the communication.
All Bits Logical 0	All bits on the transmission line are 0. In an actual implementation, this may be caused by short circuiting.
All Bits Logical 1	Analogous to "All Bits Logical 0".
Message Inversion	The bits of the message may be logically inverted. In the actual system, this may be caused by a change in the reference voltage.
Synchronization Slip	Refers to the loss of the synchronisation of receiver and sender.
Random Errors	The bits of a message may be altered in a random or non-deterministic manner.
Burst Errors	
Systematic Errors	The manifestation of repeated error patterns.

Table 5-6: Quantitative Data integrity errors

6 Qualitative Fault Modelling

Test generation always faces huge challenges, as it has to cope with all the potential faulty instances, or at least with a representative subset promising a proper fault coverage by the generated test set. Detailed automated test generation procedures are costly since they have to deal with a detailed model of the system in order to preserve its faithfulness and consequently a good correlation with the anticipated set of faults. The costs of test generation can be reduced by eliminating the generation of tests in cases when the tests would become redundant or inefficient from the point of view of covering relevant (critical) faults.

This section (based on [Pataricza, 2006] and [Pataricza, 2008]) presents a *qualitative modelling approach* at the level of faults, errors and failure modes. This approach facilitates the reuse of existing formal methods to *identify the faults that have critical effects from the point of view of the safety and dependability requirements*. This kind of model based analysis can be applied before the test generation process in order to identify the *critical faults* that have to be covered by the tests (i.e., this approach contributes to test optimization).

The use of formal methods belongs more and more to the core technologies in the design and verification of IT systems, but their use in dependability related verification is quite limited, due to the modelling and analysis complexity originating in the large number of faulty cases. Qualitative fault modelling applies abstraction, spatial and temporal compaction techniques in fault modelling to master the complexity problem (originating both in the complexity of the modelled applications and in the large number of faulty cases that has to be considered).

This chapter is organized as follows. First the challenges and the general role of abstraction are discussed. The next step is the presentation of the basic model covering faults, errors and failures. In the subsequent sections spatial compaction, temporal compaction, dynamic and static modelling techniques are presented. Finally, the role of qualitative fault modelling in the development workflow is discussed.

6.1 The Role of Abstraction in Modelling

The main advantage of formal analysis methods is their exhaustiveness, as they examine the system thoroughly by checking all potential cases. One of the major risks related to the use of formal methods is their ‘all or nothing’ nature: if they terminate, then due to the exhaustive nature of the checks they deliver certainly correct and complete results, however, if the problem complexity exceeds a limit, they deliver no result at all. This way the practical use of formal verification techniques is the strong limitation on model complexity that limits the feasibility of the underlying computations. Thus, either the faithfulness of the model has to be sacrificed by a high level of abstraction, or a proper workaround has to be found reducing the complexity by decomposing the analysis problem into small fragments that are feasible to solve.

6.1.1 The General Concept of Qualitative Abstraction

Knowing all the difficulties in analysis techniques even dealing only with the fault-free instance of the system under verification, an exhaustive formal analysis of all *potential faulty instances of the system* exacerbates the complexity problem.

A recurring motive in the related research is the search for *good abstractions* which are faithful enough to deliver practically meaningful results but fit to the computational complexity limitations of formal analysis methods.

In practice, engineers frequently use *qualitative abstractions* in reasoning about operations of a system. For instance, the formulation of the simple statement “if the temperature is higher in a technological process than a given limit then intervention is needed” uses this technique in an intuitive way. The main characteristic of the abstraction technique in the background is that all values belonging to a domain representing a similar condition are aggregated into a single representative state. (If details of the intervention are taken into account then the domain may have to be split into two or more sub-domains or a non-deterministic model has to be created.)

Note that this abstraction corresponds to a complete uniformisation of the behaviour of all the concrete states in the particular domain by allowing in the abstract model a behaviour for *all* the elements in a domain, what *at least one* element can do. This abstraction technique is known as *existential abstraction* [Clarke, 1994]. This permissive over-approximation covers all the potential behaviours of the system guaranteeing that no important case will be neglected. This way a proper abstraction technique assures a worst case analysis by assuring that no (safety) critical case is omitted from the further examination. On the other hand, over-approximation may introduce so-called *spurious solutions, i.e. they may identify potentially critical situations which have no counterpart in the implementation and originate solely in the mathematical abstraction. Similarly to false alarms in technical systems, spurious solutions may drastically reduce the efficiency of test generation*.

This way, the abstract system leads to a semi-decision. If a check over the abstract model delivers a negative result indicating that no behaviour having some target property exists (for instance there is no safety critical situation), then this is a proof of non-existence of such a behaviour in the concrete model. If a counterexample is found, further checks are needed over a refined abstract model or over the concrete model to decide whether it is a spurious solution or a genuine behaviour.

In general, abstract models generated by qualitative abstraction from of the full (concrete) model of the system represent the modes of operations inducing qualitatively different behaviours by a few values marking these operation domains and provide an over-approximation of the potential behaviours of the system. The drastic aggregation of domains into single states avoids the traditional computational complexity problems originating in the over-detailed representations in full models at a price of introducing potentially spurious solutions.

6.1.2 Qualitative Abstraction in Fault Modelling

The general principle of qualitative abstraction can be used in fault modelling based test generation, as well. The basic approach covers the *fault-free* and a *faulty* model simultaneously, in a similar form as traditional gate-level automated test pattern generation (ATPG) does it since more than four decades.

Qualitative abstractions in fault analysis describe the information flow in a potentially faulty system only at the level of resolution of fault, error, and failure modes. Here the selection of these modes is a free design parameter according to the required level of diagnostic resolution. The information contained in the simplest good/bad model is just sufficient to decide whether the individual elements and signals are good or erroneous (or what kind of error mode is present). The abstraction avoids the use of the corresponding concrete values, which would necessitate the handling of huge state spaces. Note, that As the selection of fault/error/failure modes is a free design parameter in modelling, the analyst can refine the model according to the designated test/diagnostic resolution after identifying at the highest level of abstraction the potential problem subject to more detailed checks (decision whether the particular problem is a feasible problem or a spurious solution).

The underlying core engineering heuristics is that errors showing a similar behaviour (belonging to the domain of a particular error mode) in a physical system typically cause similar impacts, accordingly, operation domains corresponding to the individual error modes can be aggregated into a single state representing the entire domain. As mentioned, such a reduction of model complexity (typically of many orders of magnitude) has the price of introducing spurious results that need further investigation (to check whether they represent a real fault effect).

The research was focused on two strongly correlated fields [Pataricza, 2006]:

- Static, syndrome-level[2] models describe the signal flow in a system after a *temporal compaction* mapping complete error sequences into a single attribute of failure mode. The set of failure modes contains in the simplest case of pass/fail categorization only the values of *good* and *faulty.* In the case of a more detailed modelling the faulty case is refined into multiple values according to *failure modes, for instance by assigning to each error sequence the most severe error value occurring in it*. The core idea of syndrome level analysis is the characterization of the sensitivity of the individual components in the system by means of relations between the error manifestations at their inputs and outputs.
- Another option, *dynamic error propagation analysis* retains more details of the dynamics of propagation of discrepancies appearing as fault impacts in a system by creating error propagation automata out of the dynamic model of the fault free system and the anticipated faults in it. These models can be created heuristically, or systematically from the dynamic description of the components.

6.2 The Basic Model

We assume that the model of the system under evaluation is available as a network of interconnected (functional or physical) components according the usual model-driven design practice. The main objective is to create the system model by synthesizing it from component models, where the component models can be

[2] A syndrome is the recognisable sign of an error (difference from the fault-free behaviour). Syndromes may aggregate data domains or data sequences into a single value.

individually and separately constructed (and potentially stored in a library of models). The approach working at the level of smaller components allows for using relatively expensive methods in the generation process of their respective dependability models (for instance by creating faulty component description libraries).

The consideration of faults necessitates the modelling of two basic phenomena: the local effects of faults at the fault location, and the propagation of errors (discrepancies) across the system in order to estimate, which faults may cause a failure.

The system model is composed as a structure of interconnected components. A *set of anticipated faults* is associated to the individual components. Each component has a fault free *reference instance* and a separate *faulty mutation* is created for each anticipated *fault mode* describing the behaviour in the presence of this particular fault.

The notion of *error* refers to impacts of faults observable as a discrepancy in the state of the actual component from that in the reference component. Accordingly, the analysis of error propagation necessitates the tracing of information flow in two concrete models: in the *reference model* and the *actual model,* both characterized by their state spaces, initial states and state transition relations. *Composite states* are defined as the (reference/actual) value pairs at the corresponding nodes in the reference and actual models.

6.2.1 Example: Car Alarm System

To facilitate the understanding of the concepts and methods described in the following, we introduce a car security application as a running example. The car security application monitors the various sensors (infra red, ultra sound, door opening, etc.) and triggers the alarm if the sensors detect any malicious activity. The alarm is triggered only if the application is armed.

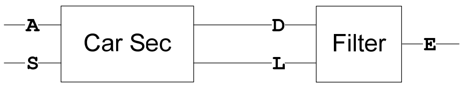


Figure 6-1: Component network of the car security system application.

Our implementation is built up from two components: the car security main unit (*Car Sec*) and a Filter component. The component network of the system is shown in Figure 6-1. The input signals of the Car Sec unit are the *arm A*, which triggers the arming and unarming of the system, and the *sensor S*, that represents the input from the sensors. The outputs are the *display D*, which indicates the armed state of the system, and the *alarm L* which is on if the alarm should be sounded and off otherwise. The outputs of the Car Sec module are used as inputs for the Filter that produces the *emergency E* signal if the system is armed and the alarm is sounded. The signals are summarized in Table 6-1.

Signal	**Type**	**Comments**
A (arm)	on, off	on: arm the system off: unarm the system
S (sensor)	on, off	on: detection of an intrusion off: no detection of an intrusion
D (display)	on, off	on: system armed off: system unarmed
L (alarm)	on, off	on: alarm triggered off: alarm inactive
E (emergency)	on, off	on: emergency situation (probable intrusion) off: no emergency

Table 6-1: Signals in the security system.

Our analysis aims at identifying the possible ways how an error may occur based on an anticipated set of faults and how it is propagated through the system. The following especially focuses on the Car Sec unit.

Reference model of the Car Sec component. The reference behaviour of the Car Sec unit is depicted in the left part of Figure 6-2. In short, after turned on, the component starts listening on sensor signals, and triggers an alarm signal in the case of an intrusion detection. If turned off, the alarm is quiesced and the component stops listening on sensor signals. The initial state is the Off state and the labels on the arcs are the respective triggers of the given transitions following the usual state chart notation, i.e. *A* and *S* mean the *on* values, while ¬*A* and ¬*S* mean the *off* values on the corresponding inputs. For simplicity, the state machine was composed according to the Moore model, thus the outputs are determined only by the current state. The output is coded similarly to the input values in the labels of the states.

Stuck-at-on fault of the Car Sec component. The Car Sec unit can be faulty in many ways. In this example we model the case when the unit cannot be turned off once it was turned on. The state machine of this faulty behaviour is depicted in the right part of Figure 6-2. The faulty state machine was created by changing all ¬*A* transitions from the Armed and Alarm states to the Off state to self-loops, thus, preventing the faulty system from turning off.

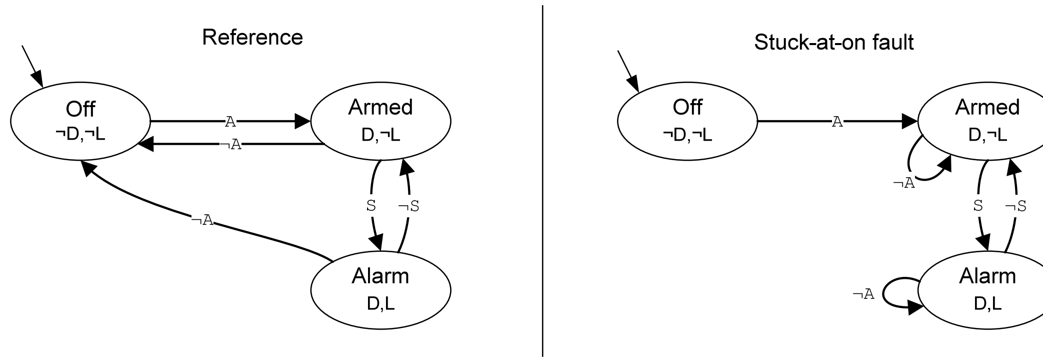


Figure 6-2: Reference and faulty state machines of the Car Sec unit.

Creating the composite automaton. The altered behaviour of the faulty model may result in deviations from the reference behaviour which are potentially observable on the outputs as errors. Figure 6-3 depicts the overview of the *composite automaton* of the Car Sec unit in which the fault-free reference and the potentially faulty actual models are composed into one product model to be used for tracing the impact of deviances. In our case it will be the *no turn off* behaviour. The compositions enable the detection of deviances that the faults in the actual implementation introduce and allow the determination the error propagation properties of the actual system. The inputs and outputs of the composite model are composite signals (signal-pairs). The labels indicate the composition with <value>,<value> pairs. The first value is the input of the reference automaton, and the second value is the input of the actual automaton. The composite output signals are defined likewise.

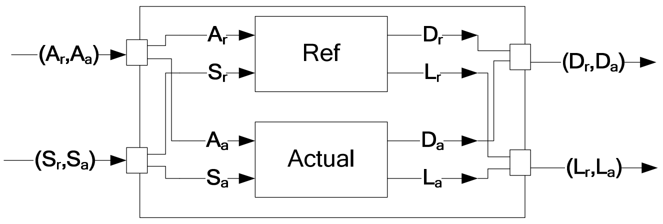


Figure 6-3: Composition of the reference and the actual automaton.

The stuck-at-on composite model. The composite model for the *stuck-at-on* fault mode is shown in Figure 6-4. The product automaton has nine states corresponding to all the possible combinations of the states in the reference and the faulty automaton. The labels of the states are <reference state>/<actual state> tuples corresponding to the current state of the reference and the actual model. Transitions are created according to the general rules of composing product automata. A transition between a state and its successor is present in the composite model for a combination of input values if corresponding transitions exist in the original models for the given input. The labels of the transitions describe all the input value combinations that trigger the transition. In the current case the composite automaton is deterministic, i.e. one combination of inputs triggers at most one transition from the current state.

Figure 2-4 contains only those transitions where the inputs of the reference and the actual models are the same. These are the *good inputs*. This corresponds to the description of the case where a faulty component is embedded into an open loop environment and the purpose of test generation is fault sensitisation, i.e., the generation of such an input sequence, which result in fault manifestation by making an error observable at the output of the components.

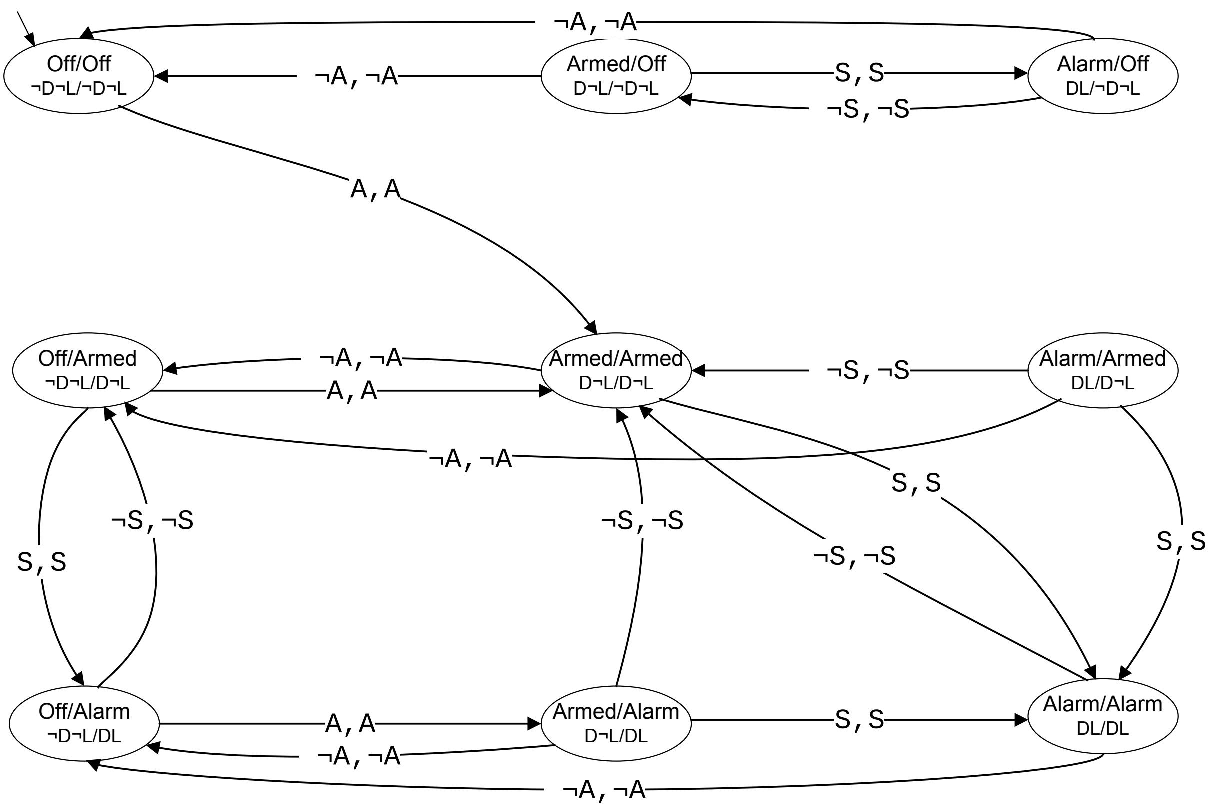


Figure 6-4: Car Sec unit composite automaton with fault free inputs

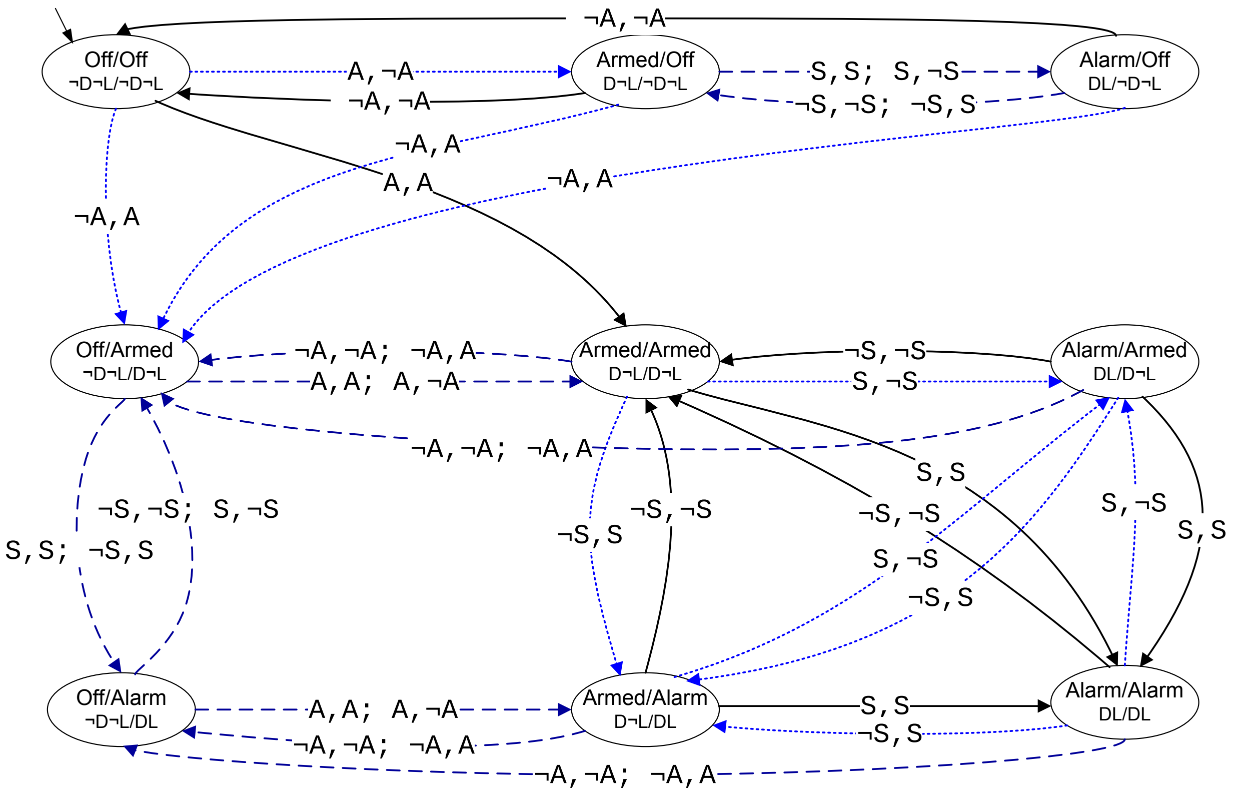


Figure 6-5: Car Sec unit composite automaton with possible faulty inputs

However, it is possible that even the input contains an error, in which case the inputs to the reference and the actual models will be different. Such a situation may occur if the component under evaluation is driven directly or indirectly by another faulty component or if the system has a feedback loop incorporating the faulty component under evaluation. Figure 6-5 shows the composite automaton extended with the *erroneous inputs*. The transitions are classified into three categories: only good inputs (solid line), good and erroneous inputs (dashed line), and only erroneous inputs (dotted line).

The reference model of the Filter component. The Filter component is a stateless logical AND gate. The *emergency* output is produced as the logical AND of the *display* and *alarm* signals, that is, the *emergency=on* iff *display=on* and *alarm=on*.

The subsequent discussion illustrates by means of this simple example the methodology of assessing the impact of component faults on the behaviour of the system embedding it. The final objective is to determine the error propagation attributes of the application. This information can later be used for answering questions like "What types of input errors are propagated by the system?"

6.3 The Concept of Signal-level Spatial Compaction

Mutually exclusive e*rror predicates* chosen by the designer partition the set of composite signals into disjoint subsets and map them to the set of *error modes*. Error modes are a spatial abstraction of the concrete value pairs appearing at the corresponding nodes.

The simplest classification reflects only the *match* and *mismatch* of the values in a composite signal by taking the error predicate as the equality (=) and inequality (≠) relation. It partitions errors into the error modes of {*good*, *faulty*} and neglects the particular type of the deviance (see the left part in Figure 6-6). More fine grained models differentiate between error modes by separating the class of *faulty* into disjoint subclasses, like differentiating *minor* and *major deviance*s and *out of range* values (see the right part in Figure 6-6).

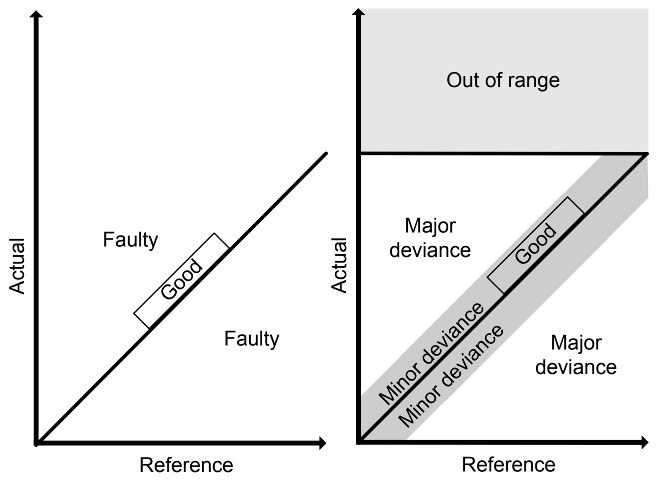


Figure 6-6: Refinement of the error mode

Naturally, more complex error mode representation may introduce values depending on the reference and actual states as well in addition to the deviance itself. The formulation of error predicates is an important instrument to reflect the actual dependability objective.

Even in the case if a signal is a pure binary one, the domain refinement of the faulty case may be of interest as it is shown below in the running example.

6.3.1 Example: Definition of Error Modes and Error Symbols of Signals

Errors appear as discrepancies between the data values of the reference and the actual models. As all the signals in our example application are binary ones, all input and output ports in the application can take one of the *on (1)* or *off (0)* values. The port value assignments in the composite model appear in <reference,actual> pairs and the relation of the values defines the specific *error modes*.

The selection of the resolution of the categorization of composite values is a free design parameter. The candidate choices are depicted in Figure 6-7. In the case of the concrete model all the four value pairs are represented by separate values[3], thus no data domain aggregation and abstraction is employed. The fine

[3] In the specific case of pure binary signals, this corresponds to the Boolean algebraic values of 0,1,D, ¬D used in the traditional gate level logic testing.

granular model uses an aggregated representation of the good values but keeps separate representation for the two bad cases. In go/nogo modelling only the classes of good and bad are distinguished and the values within a class are aggregated into a single class.

Note, that in a system model different granularities of signal value abstractions can be used.

A guideline for selecting the signal abstraction level is retaining the criticality of their impact. In our case, stuck-at-inactive faults may generate a complete violation of the protection, while stuck-at-active results in the less severe false alarms. Due to the very different potential impact it is worth to keep them as separate classes in the model.

In our simple running example, such a distinction is straightforward, but in complex safety cases the proper level needs a more systematic approach.

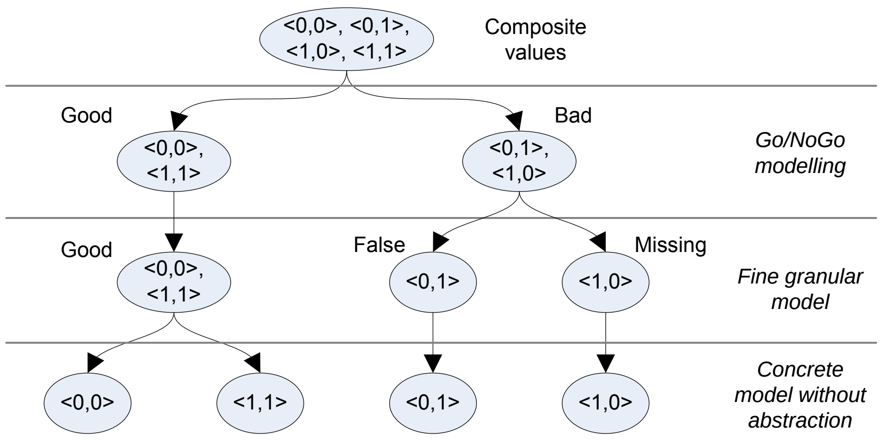


Figure 6-7: Resolution of signal error modes

The relations can be classified into three categories:

- `Ok`, when the values are equal (*reference=actual*) {<0,0>;<1,1>}
- `False`, when the actual is falsely on (*reference=off* $\wedge$ *actual=on*) {<0,1>}
- `Missing`, when the actual is falsely off (*reference=on* $\wedge$ *actual=off*) {<1,0>}

These error modes appear as *error symbols* on the ports of the composite automaton, for example for input *A*:

- E_{A0}: Ok <A_{ref}=0, A_{act}=0>, <A_{ref}=1, A_{act}=1>
- E_{A1}: False <A_{ref}=0, A_{act}=1>
- E_{A2}: Missing <A_{ref}=1, A_{act}=0>

Error symbols are defined similarly for all inputs and outputs.

Even in our very simple running example having only four binary signals the spatial compaction reduced the state space of the composite signals to be explored from $4^4 = 256$ to $3^4 = 81$. This could be further reduced to $2^4 = 16$ if the distinction between *false* and *missing* errors is not necessary. The price of the reduction is the introduction of spurious transition sequences by the existential abstraction used. For instance, in the abstract Go/NoGo type model two subsequent steps are allowed even if the individual steps refer to different (mutually exclusive) error modes. This way the practical use of the abstraction needs a careful trade-off between state space reduction (low resolution) and keeping the number of spurious solutions low necessitating a high resolution. Although this is not a considerable reduction since signals are binary, in data rich signals the compaction of value domains into single qualitative values result in huge complexity reduction.

6.4 Dynamic Modelling

Abstract dynamic error propagation analysis relies on co-modelling of the dynamics of the target reference and actual systems (i.e., internal operation sequences in the components, their mutual interaction and invocation) in order to incorporate the activation sequences into the analysis. This dynamic model describes the behaviour of the reference-actual model pair after spatial compaction by using the compacted qualitative values as inputs/outputs.

The introduction of error modes facilitates the creation of a simple automaton describing the impacts of errors appearing at inputs of the individual components. An input error may result in

- an error on the output of the corresponding component (error propagation),
- a discrepancy in the states in the reference and actual models (e.g., a latent error potentially manifested later as a control flow distortion), or both,
- an error absorption (e.g., a single input error of a TMR component).

This dynamic qualitative error propagation modelling automaton uses the error modes as input-output values to characterize the behaviour of the composite model. An *error sequence* is a temporal sequence of discrepancies represented by their respective error modes.

The principle of construction of the abstract automaton is a special form of *predicate abstraction* [Graf, 1997]. The core observation behind this abstraction paradigm is, that the branching predicates in a program partition the state space into disjoint operation domains, and all data within a particular domain behave similarly. This way the representation of a data rich algorithm can be substituted by a skeleton one tracing only the execution trajectory with the resolution of operation domains instead of the detailed data values within them. This technique assigns Boolean predicate values as independent variables to the control branching predicates over the detailed data values of the system. For instance the branch selection in an IF <condition> THEN ... ELSE ... statement will be substituted by an independent Boolean variable. The executable value assignments in the THEN and ELSE branches will be mapped in the abstract model into logic expressions describing the value changes of all Boolean variables representing the predicate valuations, Using existential abstraction, one can obtain a corresponding abstract transition system.

In our methodology for fault modelling a state transition will be included into the *abstract* state automaton from the actual composite state to a given successor state upon an input error vector and generating an output error vector, if there exists at least one arbitrary combination of input and output vectors, and actual and next states in the *concrete* model corresponding to it. This simplest form of model construction necessitates a satisfiability check for each combination of input/output vectors and actual states.

This model represents the control flow in the abstract model in its full extent, but the representation of data dependencies is reduced to that on syndrome values thus aggregating entire data domains into a single syndrome value.

The abstract automaton can answer for instance, the typical question for the impact of the first occurrence of a "minor" error. Here, the initial states in both the actual and reference concrete models are identical and the question is "*Over an arbitrary pair of input values differing only slightly at the corresponding inputs of the actual and reference models what are their successor states and what kind of deviance may appear on their outputs?*". Such a question may help to conclude whether minor deviances can cause divergences in the control flow executions (different runs) of the actual and reference systems. Similarly, it can be checked whether such an input of a minor deviance may be amplified to a major one on the outputs. Both answers are approximate in the sense that they indicate only the potential occurrence of such a safety problem.

The creation of an automaton working over the few values from the error modes as input and output alphabets requires exhaustive checking of all value pairs constrained by the error mode for all the reachable state pairs. This is one of the core elements in the algorithms that requires computational power; however, for simple components this can be done even by hand.

One important characteristics of the dynamic modelling approach sketched above is that it uses only very basic constraints on the modelling language as the concepts used are confined to discrete state space and time, non-deterministic finite automata. This way, popular modelling approaches like dataflow, sequence or statechart diagrams can be extracted in such a way that the abstract model uses the same modelling paradigm. All the algorithms used for checking the particular language (used in describing the original model) can be re-used without any alteration to check fault effects.

The system model is created by means of composing the individual component models to a network. Thanks to the drastic reduction in data representation, the computational complexity for analyzing this model lies in between that related to the non-interpreted model (neglecting data values) and proof of correctness models which are rich in data but treat only a single, good system.

The advantage of the spatial compaction is that the abstract automaton has to operate over the very few discrete values corresponding to the error modes even in the case if the system is continuous or hybrid. The complexity of the state spaces corresponds to the Cartesian product of the state spaces in the reference and actual models.

6.4.1 Example: Dynamic Modelling

The dynamic model of the car alarm system allows us to analyse the deviances that may be present between the actual implementation and the reference behaviour model.

The dynamic model is created based on the composite automaton (Figure 6-8). The states of the dynamic model are the same as that of the composite model. A transition exists in the dynamic model with an input error symbol as trigger if a corresponding transition exists in the composite model with a composite signal as trigger, where the composite signal is transformed into the given error symbol during spatial compaction.

The output signal pairs are also substituted with the appropriate output error symbols in the dynamic model.

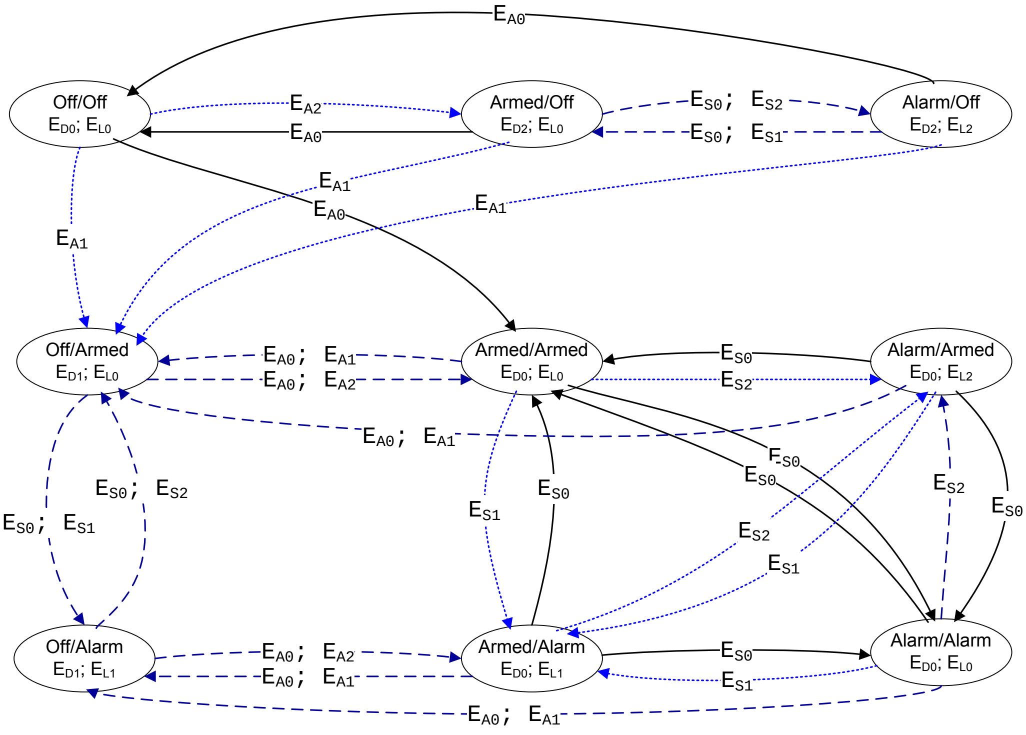


Figure 6-8: Dynamic model of the Car Sec composite automaton

6.5 The Concept of Temporal Compaction

The spatial compaction method achieves a huge complexity reduction by aggregating large data domains into a few qualitative values. Another option for further reduction of computational complexity is along the *temporal dimension* by compacting arbitrarily long error sequences into a single value (syndrome). In engineering, it is common to use such simplified views of long sequences. For instance, software errors are frequently characterized (at the level of resolution of severity class) only according to the most severe error occurring in them, independently of the value and the temporal position of the error occurrence.

The most convenient way to express temporal attributes is by using *temporal logic*, a formal logic widely used to specify and reasoning about logic propositions qualified in terms of time. A complete and mutually exclusive set of predicates formulated as temporal logic expressions performs a temporal abstraction by partitioning the set of error sequences into disjoint subsets and compacting each of them into a single element from the set of *syndrome* values. Syndromes observable by the system user are referred to as *failure modes*. The selection of the syndrome value set is a free design choice offered to the analyst according to the purpose of the analysis, similarly to error modes. For instance, a set of syndrome values in fault impact analysis may contain the values of *{good, minor failure, major failure}* ordered according to the severity of the fault impacts. In the temporal domain values like *{right in time, too late}* can be considered.

Syndrome level modelling constitutes the topmost level of abstraction in error propagation modelling. The principle of creating syndrome level component models is similar to the estimation of dynamic ones. A component syndrome model describes the sensitivity of its outputs to failure modes appearing on its inputs in the form of a set of input-output syndrome relations. For instance, a syndrome relation describes that if a <*good, minor failure*> combination of inputs is received by a given component then it produces a *minor failure* at its output.

The estimation of a particular syndrome relation needs the checking for each input-output value vector combination whether there exist such error sequences which fulfil the candidate relation. This necessitates for each value combination a model checker run. The resulting computational complexity confines the maximum complexity of the component models to a moderate level.

In the following the syndrome relations that are proved by the model checker are called *propagation rules*. These rules show the possible input-output relations of, e.g. whether the good input can result in erroneous output. Later these rules will enable the estimation of the possible error propagation rules of the complete system without computing the dynamic model of the complete system, thus, possibly decreasing the complexity of this task with orders of magnitude.

6.5.1 Example: applying temporal compaction

In our case we use Linear Temporal Logic expressions to describe the syndromes on the ports of the car alarm system. The expressions need to be defined for all inputs and outputs as a complete and mutually exclusive set.

The syndromes defined for the Car Sec component are listed in Table 6-2.

Input/Output	Syndrome	Description	LTL expression
arm input syndromes			
	A0	OK	$G(E_{A0})$
	A1	False arm	$F(E_{A1}) \wedge \neg F(E_{A2})$
	A2	Missing but not false arm	$F(E_{A2})$
sensor input syndromes			
	S0	OK	$G(E_{S0})$
	S1	False sensor	$F(E_{S1}) \wedge \neg F(E_{S2})$
	S2	Missing but not false sensor	$F(E_{S2})$
alarm output syndromes			
	L0	OK	$G(E_{L0})$
	L1	False but not missing alarm	$F(E_{L1}) \wedge \neg F(E_{L2})$
	L2	Missing alarm	$F(E_{L2})$
display output syndromes			
	D0	OK	$G(E_{D0})$
	D1	False but not missing display	$F(E_{D1}) \wedge \neg F(E_{D2})$
	D2	Missing display	$F(E_{D2})$

Table 6-2: Syndromes of the Car Sec component.

The *alarm* and *display* outputs are used as inputs for the Filter component. The output syndromes for the *emergency* output port are listed in Table 6-3.

Input/Output	Syndrome	Description	LTL expression
emergency output syndromes			
	E0	OK	$G(E_{E0})$
	E1	False emergency	$F(E_{E1}) \wedge \neg F(E_{E2})$
	E2	Missing but not false emergency	$F(E_{E2})$

Table 6-3: Syndromes of the AND logical gate.

All combinations of these syndromes (*syndrome relations*) need to be evaluated on the dynamic model of the corresponding component and the ones that can be proved are the possible *(static) propagation rules*. The calculation for a component can be carried out with automatic methods or it can be done manually. The propagation rules of the Car Sec component will be calculated by the automatic method while the Filter component will be examined manually.

Automatic syndrome relation computation. The automatic method we implemented uses the symbolic model checker tool of the Symbolic Analysis Laboratory [Bensalem et al, 2000] to prove the syndrome relations[4]. The models are implemented as SAL modules and the syndrome relations are described as Theorems to be proved. The format of the theorems is:

$$\neg\left(\prod \varphi_i \wedge \prod \psi_i\right)$$

[4] The method is a proof-of-concept implementation of the automatic computation of propagation rules.

where φ_i are the input syndromes and ψ_i are the output. In other words, the formula is the negated conjunction of all input and output syndromes. If a theorem proves to be *invalid*, then it means that the formula itself is valid. The reason for using the negated form is that the model checker tool can provide a counter example to the invalid theorems, which is an actual trace in the model where the formula is valid. The counter example can be used to check the sanity of the model and to check if the syndromes really express what they should.

The calculation for the Car Sec component selects the valid propagation rules from the 81 possible ones. Specifically, in the case of good inputs (A0 and S0), the method proved only 3 of the 9 possible propagation rules. A few examples of these 9 rules are listed in Table 6-4.

Syndrome relation	**Validity**	**Interpretation**
$A0 \wedge S0 \wedge L0 \wedge D0$	valid	If all inputs are good, all outputs can be good.
$A0 \wedge S0 \wedge L0 \wedge D1$	valid	If all inputs are good, the *false display* syndrome can be observed on the output while the *alarm is still good*.
$A0 \wedge S0 \wedge L2 \wedge D0$	invalid	It is impossible that on good input an alarm is missed. This is in concurrence with our previous statement in Section 6.4.1: the system cannot go into the Unarmed or Missed alarm states if the input is good.

Table 6-4: A selection of propagation rules of the Car Sec component.

Manual syndrome relation computation. For very small components, like the Filter component, which is actually an AND logical gate, the manual analysis may also work. The AND gate receives the *display* and the *alarm* so we need to check how does it propagate the symbols that these outputs can emit. Table 6-5 shows the values that can appear on the *D* and *L* outputs. The values are shown in <value>/<value> tuples, where the first value is emitted by the reference model and the second value is emitted by the actual model. The domain of the D and L outputs is {*off, on*} which is translated to {0,1} values in the table to ease the evaluation of the AND function on them.

ref/act	**Ok**		**Missing**	**False**
D	0/0	1/1	1/0	0/1
L	0/0	1/1	1/0	0/1

Table 6-5: Values of the error symbols of the D and L outputs.

The composite model for the AND gate is depicted in Figure 6-9. The actual model is equivalent to the refrerence (the AND logical function) but it receives the inputs from the actual model of the previous component (D_{act} and L_{act}). The input is evaluated on both models and the *E* output is calculated as the relation of the E_{ref} and E_{act} outputs. The E output symbols are calculated as the relation of these two values, and again results in the {*ok, missing, false*} symbols as defined in Section 6.4.1.

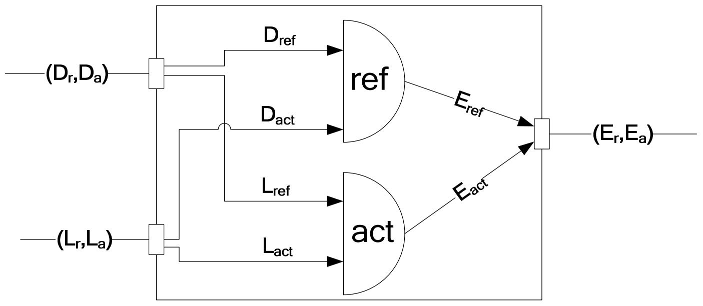


Figure 6-9: The composite model of the AND gate.

Let us take an example scenario. The input symbols are: D=ok, L=missing. The evaluated functions are $E_{ref}= D_{ref} \wedge L_{ref}$ and $E_{act}= D_{act} \wedge L_{act}$. The input values are translated as: $<D_{ref},D_{act}>=\{<0,0>;<1,1>\}$, $<L_{ref},L_{act}>=<1,0>$. Evaluating the functions the results are: $<E_{ref},E_{act}>=\{<0,0>;<1,0>\}$. Translating this to error symbols results in the {ok, missing} set of possible outputs and the corresponding {E0, E1} syndrome set. This propagation rule is highlighted in Table 6-6. In addition, the table lists all the propagation rules of the AND logical gate. The syndromes are substituted by the first letter of their canonical name for better readability.

L	D	E
o	o	o
m	o	o/m
f	o	o/f
o	m	o/m
m	m	o/m/f
f	m	o/f
o	f	o/f
m	f	o/f
f	f	o/f

Table 6-6: Propagation rules of the AND logical gate.

Summary. The final outcome of the temporal compaction is the set of propagation rules for each component. If more of the same component exist in a system, then the same rules apply to all instances.

6.6 Syndrome Level Static Modelling

Syndrome level static models (Figure 6-10) composed of interconnected failure relation models of components use a high level of abstraction *both in the spatial and temporal dimensions* representing their mutual interdependency.

The formerly introduced models represent the component types, as they describe the general behaviors of the components. These component types are used as the definition of the components, and then we can compose the syndrome level static model from the individual component models.

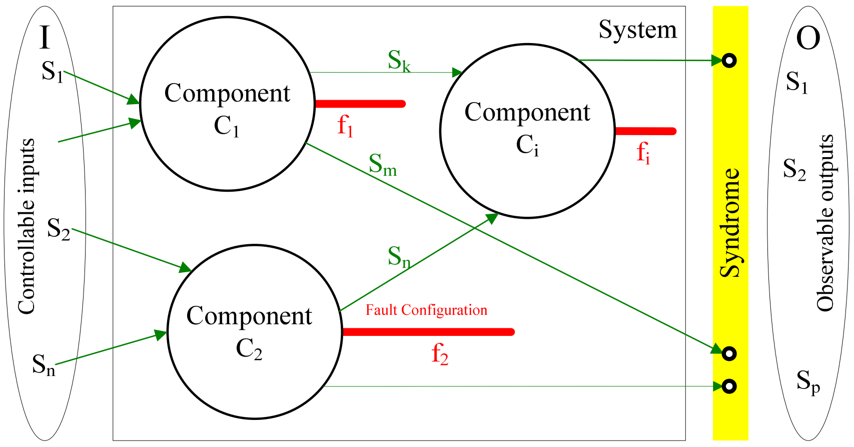


Figure 6-10: Syndrome-level static network

Syndrome level static models are composed as a network of relations and the problems described by such networks are referred to as Constraint Satisfaction Problems (CSP). CSPs are networks of variables interconnected by a set of relations called constraints. Here constraints represent the input-output syndrome relations. During testability analysis additional constraints may represent the need that (in case of testable faults) the observable system outputs shall be distinguishable from the correct outputs.

A CSP is solved, if a value assignment has been found for all variables, such that all the relations are satisfied. Constraints have several interesting properties; for example a constraint does not need a unique specification of all the values of its variables, thus they may specify *partial information*. Constraints are non-directional, declarative (they specify what relationship must hold without specifying a computational procedure to enforce that relationship), additive (the conjunction of constraints is effectively independent of the order of imposition of constraints), and compositional, as hierarchical refinement is supported.

The use of relations in CSPs provides a proper support for handling nondeterministic abstractions of complex systems. For the analysis of typical architectural redundancy patterns, like n-out-of-m redundancy structure can be easily modelled in a general form. The analysis can prove the appropriateness of built-in fault-tolerance of safety critical functionalities in the case of structural redundancy at the top-most level of abstraction without necessitating a detailed model of the individual functionalities.

Static, syndrome level model based analysis delivers frequently overly pessimistic results by taking all the topologically feasible error propagation paths as potentially active, independently whether an actual scenario uses them at all.

6.6.1 Analysis of the system as a Constraint Satisfaction Problem

6.6.1.1 The CSP defined

Definition: Constraint Satisfaction Problem (CSP) is defined by a set of **variables**: $X_1, X_2, \dots X_3$, and a set of **constraints**: $C_1, C_2, \dots, C_3$. Each variable X_i has a nonempty domain D_i of possible values. Each constraint C_i involves some subset of the variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an **assignment** of values to some or all of the variables, $\{X_i = v_i, X_n = v_n, \dots\}$. An assignment that does not violate any constraints is called a **consistent** or legal assignment. A complete assignment is one in which every variable is involved, and a **solution** to a CSP is a complete assignment that satisfies all the constraints.

A CSP is solved if a *solution* is found for the problem. Constraints have several interesting properties; for example a constraint does not need a unique specification of all the values of its variables, thus they may specify *partial information*. Constraints are non-directional, declarative (they specify what relationship must hold without specifying a computational procedure to enforce that relationship), additive (the conjunction of constraints is effectively independent of the order of imposition of constraints), and compositional, as hierarchical refinement is supported.

6.6.1.2 Relation of the CSP and the static component network model

In order to reason about the error propagation characteristics of a static component network model, we have to be able to represent *error propagation rules*, that is, simultaneous syndrome combinations on the inputs and outputs of the network. As such, analysis of the network may mean e.g. looking for such hypotheses that are compatible with a priori error propagation knowledge or checking whether a given hypothesis is contradictory with these.

Formally, let us assign a variable to each individual input/output port with the respective possible syndrome sets as their domain, and assign a variable to each individual component with the internal fault modes as their domain. This way, in general analysis tasks will either look for valuations of these variables (possibly with some of it being pre-bound) or check the validity of a given valuation-vector.

Consequently, these variables build the set of variables the analytical CSPs are defined over. Regarding the constraints, two fundamentally distinct types will always be present in the CSP formulation: propagation constraints determined by components and topological constraints.

The syndrome-level propagation rules of a component type essentially define a constraint template for said component type. As these rules form a finite relation in the mathematical sense over the input/output syndromes and the internal fault mode, this relation can be translated to a finite domain CSP constraint without modification (as those are essentially finite relations, too). As such, we can post the same relation over the respective input/output variables and internal fault mode variable for each instantiation for the component type at hand. (Note that depending on the CSP engine, for performance reasons one will want to reformulate the relation e.g. into a decision tree - however, here it can be treated as a technical detail.)

On the other hand, the connectivity of the components in the static component network imposes further topological constraints on the CSP variables. In the simplest case, where an output is connected to an input, the valuation of the variables that are mapped to these ports has to be the same - a basic binary constraint in CSP terminology.

The solutions of the CSP built in this way will be the error propagation rules of the system including a priori knowledge about the state of the component network. If more constraints are added to the CSP, those will further restrict the possible value assignments and thus return a subset of the possible solutions. This can be used to focus the search of solutions to a narrow solution set. In the extreme case, if all variables are bound, the CSP becomes a decision problem with a valid/not valid answer.

6.6.2 Car alarm system - System level analysis

In order to determine the system level error propagation attributes of the car alarm application, the static propagation rules of the components need to be transformed into constraints. Then, by adding more constraints, we are able to ask the questions that are important for us.

At first we describe the propagation rules of the two types of components.

The transformation of the propagation rules is straight forward. For example, the propagation rules of the Filter component look like as follows:

$$\begin{aligned}
&\mathrm{proprule\_Filter}(\mathrm{L},\ \mathrm{D},\ \mathrm{E}):\\
&(\mathrm{L}=\mathrm{L0}\ \wedge\ \mathrm{D}=\mathrm{D0}\ \wedge\ \mathrm{E}=\mathrm{E0})\ \vee\\
&(\mathrm{L}=\mathrm{L0}\ \wedge\ \mathrm{D}=\mathrm{D2}\ \wedge\ (\mathrm{E}=\mathrm{E0}\ \vee\ \mathrm{E}=\mathrm{E2}))\ \vee\\
&(\mathrm{L}=\mathrm{L0}\ \wedge\ \mathrm{D}=\mathrm{D1}\ \wedge\ (\mathrm{E}=\mathrm{E0}\ \vee\ \mathrm{E}=\mathrm{E1}))\ \vee\\
&(\mathrm{L}=\mathrm{L2}\ \wedge\ \mathrm{D}=\mathrm{D0}\ \wedge\ (\mathrm{E}=\mathrm{E0}\ \vee\ \mathrm{E}=\mathrm{E2}))\ \vee\\
&(\mathrm{L}=\mathrm{L2}\ \wedge\ \mathrm{D}=\mathrm{D2}\ \wedge\ (\mathrm{E}=\mathrm{E0}\ \vee\ \mathrm{E}=\mathrm{E1}\ \vee\ \mathrm{E}=\mathrm{E2}))\ \vee\\
&(\mathrm{L}=\mathrm{L2}\ \wedge\ \mathrm{D}=\mathrm{D1}\ \wedge\ (\mathrm{E}=\mathrm{E0}\ \vee\ \mathrm{E}=\mathrm{E1}))\ \vee\\
&(\mathrm{L}=\mathrm{L1}\ \wedge\ \mathrm{D}=\mathrm{D0}\ \wedge\ (\mathrm{E}=\mathrm{E0}\ \vee\ \mathrm{E}=\mathrm{E1}))\ \vee\\
&(\mathrm{L}=\mathrm{L1}\ \wedge\ \mathrm{D}=\mathrm{D2}\ \wedge\ (\mathrm{E}=\mathrm{E0}\ \vee\ \mathrm{E}=\mathrm{E1}))\ \vee\\
&(\mathrm{L}=\mathrm{L1}\ \wedge\ \mathrm{D}=\mathrm{D1}\ \wedge\ (\mathrm{E}=\mathrm{E0}\ \vee\ \mathrm{E}=\mathrm{E1}))
\end{aligned}$$

It can be seen that in this type of module we defined the relation between only the input-output ports. The rules of the Car Sec component are composed similarly into the following constraint:

$$\begin{aligned}
&\mathrm{proprule\_CarSec}(\mathrm{CS\_FM},\ \mathrm{A},\ \mathrm{S},\ \mathrm{L},\ \mathrm{D})\\
&(\mathrm{CS\_FM}=\mathrm{OK}\ \wedge\ \mathrm{A}=\mathrm{A0}\ \wedge\ \mathrm{S}=\mathrm{S0}\ \wedge\ \mathrm{L}=\mathrm{L0}\ \wedge\ \mathrm{D}=\mathrm{D0})\ \vee\\
&(\mathrm{CS\_FM}=\mathrm{NOOFF}\ \wedge\ \mathrm{A}=\mathrm{A0}\ \wedge\ \mathrm{S}=\mathrm{S0}\ \wedge\ \mathrm{L}=\mathrm{L0}\ \wedge\ \mathrm{D}=\mathrm{D0})\ \vee\\
&(\mathrm{CS\_FM}=\mathrm{NOOFF}\ \wedge\ \mathrm{A}=\mathrm{A0}\ \wedge\ \mathrm{S}=\mathrm{S0}\ \wedge\ \mathrm{L}=\mathrm{L0}\ \wedge\ \mathrm{D}=\mathrm{D1})\ \vee\\
&(\mathrm{CS\_FM}=\mathrm{NOOFF}\ \wedge\ \mathrm{A}=\mathrm{A0}\ \wedge\ \mathrm{S}=\mathrm{S0}\ \wedge\ \mathrm{L}=\mathrm{L1}\ \wedge\ \mathrm{D}=\mathrm{D1})\ \vee
\end{aligned}$$

....

This type of component has also an internal fault mode, represented with the *CS_FM* parameter of the propagation rule.

After the definition of the propagation rules of the different types of components, we are able to compose the whole system together.

In Figure 6-11 we show the components and their topological relations in the system.

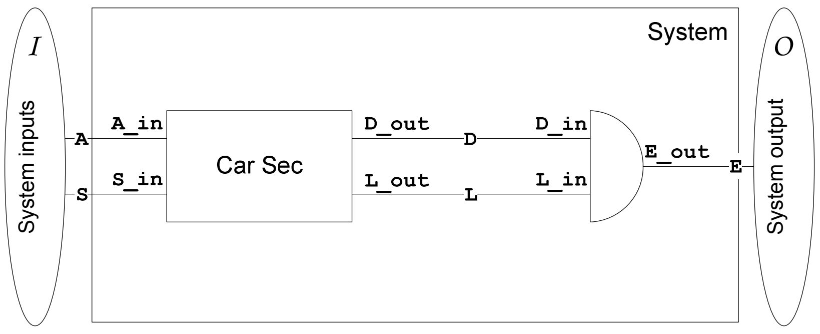


Figure 6-11: Component network and CSP variables

The constraints used to express the topological properties of the model are simple equality constraints representing the bindings between the components' outputs and inputs. In our example, the Car Sec component gets its inputs from outside of the system. These are the input parameters of the *proprule_CarSec* constraint, represented by the *A_in* and *S_in* variables. The outputs of this component are mapped to the variables *D_out* and *L_out*. Since the inputs of the Filter component are mapped to the variables *D_in* and *L_in*, we have to define the constraints representing these bindings between the corresponding inputs and outputs; in the CSP level it means equality between the variables represented by the following constraints:

$\mathrm{connectionD}(\mathrm{D\_out},\ \mathrm{D\_in})$

$(\mathrm{D\_out} = \mathrm{D\_in})$

$\mathrm{connectionL}(\mathrm{L\_out},\ \mathrm{L\_in})$

$(\mathrm{L\_out} = \mathrm{L\_in})$

Asking a question. Let us take the case when error free inputs are provided to the system. To see *what type of errors can appear on the output*, we need to add the following constraint:

$\mathrm{initial}(\mathrm{A},\ \mathrm{S})$

$(\mathrm{A{=}A0} \wedge \mathrm{S{=}S0})$

Solving the constraint problem, the following solution set is returned: $\mathrm{E\_out{=}E0} \vee \mathrm{E\_out{=}E1}$, which means in the example that $\mathrm{E\_out}{=}\mathit{OK} \vee \mathrm{E\_out}{=}\mathit{False}$

In other words, the solution for the constraint problem is that if the input is correct, the output E syndrome can be OK or False. In the real system this means that our system can:

1. Either work properly or
2. Cause a false alarm and
3. There will be no missed alarms.

6.7 Qualitative Modelling in the Development Workflow

Qualitative modelling is a trade-off between the modelling detail (and avoidance of spurious results) and the computational complexity needed for the analysis. Figure 6-12 summarizes the main possibilities for model abstraction and refinement. In the following we summarize the role of these models and the corresponding abstraction and refinement steps.

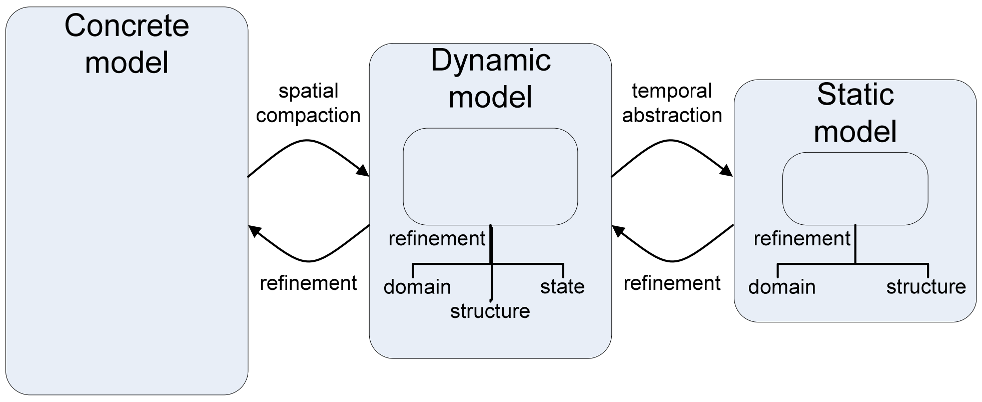


Figure 6-12: Model abstraction and refinement

6.7.1 Approaches to Construct the Dependability Model

Both dynamic and static error propagation models can be created in two ways depending on the fact whether we start from a coarse or a fine granular model:

- *Heuristic top-down modelling* serves for an early assessment of fault effects in the initial phases of system design. It may start for instance from a non-interpreted model of the control flow. The starting point is a skeletal form of the characterization of the dynamics substituting data dependencies with nondeterministic choices. This model will be enriched with dependencies on the syndrome values interpreting this way the dependence of the dynamic behaviour of the system on errors and failure modes of components. This method is intuitive, as it relies on an expert-made forecasting of the sensitivity of the individual components to input errors.
- *Abstraction based modelling* derives the dynamic error propagation models from the functional description of the components. Obviously, this model necessitates a more detailed specification of the

system than the sketch-like heuristic methodology, but it delivers an accurate estimation of error propagation effects and simultaneously assures the consistency between the functional and error propagation models.

The scope of these two models may cover either only the behaviour of the system, or it can be extended to cover both the architecture and its deployment to resources [Pataricza, 2002].

There is a difference how a model will be constructed depending whether the analysis of fault effects is executed concurrently with the design workflow or it is an *a posteriori* activity.

- In the first case the creation of the dependability model follows a step by step refinement, synchronized with the design flow of the system. It starts from a non-interpreted model of the target system representing data by their presence/absence at the different nodes of the system. The intuitive method introduces error modes by a gradual refinement into the non-interpreted model in order to get a description of the error propagation properties of the actual component instance. The subsequent refinements of the model will be executed in synchrony with the functional design workflow.

  The main drawback of the intuitive method is, that it is error prone, especially, if an input error may cause latent errors inside of a component (in its storage elements) postponing the manifestation of the error by several steps of operation. An alternate way can be followed if at each individual level the functional specification is given, in which case the dynamic and static models can be derived from the functional description by using the algorithms sketched in the previous subsections.
- Frequently, the analysis of fault effects is a separate activity following the functional design. According to our practice, the easiest way to generate good models starts from the most detailed models of simple components and creates abstract models by automated aggregation. For instance, the syndrome level static error sensitivity model of a sub-network performing a particular functionality will be generated by means of estimating of all the solutions of the corresponding constraint network and omitting from these solution relations all the variables invisible as interface signals. The basic definition of the temporal compaction described in Sect. 6.5 serves as a compliance checking criteria between dynamic and static models.

6.7.2 Counterexample Driven Model Refinement

Counterexample driven model refinement is one of the fundamental techniques in qualitative model analysis. The core idea here is that a specific refinement is performed after finding a spurious counterexample, if it has been found due to abstraction but does not apply to the target model. (In our context, the counterexample is considered from the point of view of satisfying dependability requirements, i.e., it incorporates one or more faults that have critical effects in the system.) This refinement can be performed locally around the location or temporal interval in which the problematic fault effect was detected. Here the abstract counterexample is used as a boundary constraint in the finer granular model. All the refinement possibilities illustrated in Figure 5-3 can be included in the model based analysis process in order to support the elimination of spurious solutions (counterexamples) originating in the over-abstraction characteristic of the abstract models.

- *Domain refinement* uses a finer resolution in the error/syndrome modelling or introduces a more refined separation of cases into the control flow. Input-output refinement introduces sub-types to error modes/syndromes.
- *Control flow refinement* decomposes the state space of a component by splitting an internal state into multiple ones and simultaneously modifying it. The set of state transition relations is in order to assure the correspondence between the states in the original and refined models.
- *Structure refinement* splits a single component into a sub-network composed of multiple components, while the interfaces and the state space remain unaltered. Changes of the network structure effect only the ”inner structure” of the refined node, but the other nodes as well as the original interconnections remain unaltered.
- *State refinement* increases the temporal resolution of the model by splitting state transitions into a sequence of transitions and introducing additional sub-states. For instance, an initial coarse model of an operation describes only the arrival of the request to the corresponding unit and the response given by it while a more fine granular one includes the internal control flow of the operation as well.

6.8 Summary

In this chapter we investigated the application of qualitative fault modelling techniques to address the complexity problem that often occurs when the fault effects are analyzed in complex systems consisting of several interconnected components. We introduced a temporal and spatial abstraction based methodology that offers several abstraction possibilities. Thanks to the high level of information compaction, the abstract solution space is of a feasible complexity. Abstract models can be utilized according to the following general rules and conditions:

- By using over-abstractions *the abstract model delivers an upper cover of the feasible solution space*. This way there exists no solution outside of the concrete counterparts of the abstract solution space.
- A semidecision procedure can be built to determine the relevance of a solution in the concrete domain: if a behaviour in the concrete domain has no counterpart in the abstract solution space, we can certainly exclude it; however, if it is included, we cannot decide whether it is a true behaviour or only a spurious solution originating in the abstraction technique used.

We proposed these qualitative abstraction techniques for the purpose of *analyzing fault effects*, i.e., to identify faults that may have critical effects from the point of view of safety or dependability of the application. This formal analysis of fault effects is advised to precede test generation, since this analysis *allows identifying the critical faults that have to be in the focus of the test generation*. Moreover, testability properties can also be derived (i.e., checking whether a given fault can be tested by considering the points of control and observation in the system). Note that besides this particular application, there are also other potential application areas of qualitative fault modelling that would need further research:

- In the case of mutation based testing, the abstract counterparts of the mutations (mutant models) could be generated. Solving the test generation problem in the abstract space, the abstract solutions can be used as a guiding heuristics in the concrete detailed (SAT based) test generation: here the core idea is to add the abstract solutions – together with the mapping to concrete domain – as additional conditions to the satisfiability problem to be solved. It is a hierarchical approach, since the abstract model deals with reducing the search space of the detailed concrete model by eliminating all those solutions at the high level which are irrelevant.
- In the case of *fault injection* the main purpose of using the abstract models could be the pre-filtering of the patterns used in the fault injection campaigns. On the one hand, coverage can be estimated (already evaluated failures can be eliminated from the injection), on the other hand, uncovered failures can be estimated together with their test sets (aiming at a better focusing of fault injection campaigns).

The main advantage of the proposed modelling technique is the reusability of existing formal methods for analysing fault effects. From an implementation point of view, *static analysis* can be performed by means of constraint satisfaction programming. The analysis can be further refined by applying the costly but more accurate *dynamic analysis*. Both methods are characterized by the use of over-abstraction of the system thus they guarantee that *no critical behaviour will be overlooked*. This way, they fulfil the expectations against a worst case analysis. The price of the simplified analysis is that (depending on the level of simplification) they may generate spurious results that can be eliminated by a subsequent detailed analysis.

In the preceding subsections we demonstrated the potentialities of the methodology in an example. In the current status of implementation, the fault modelling and the proposed temporal and spatial abstraction techniques need several systematic but manual steps. The development of fully automated tool support is a challenging but secondary activity that is beyond the primary scope (and resources) of the project.

7 Conclusion

"The fault models catalogued in chapter 5 are by far going beyond those related to the industrial experiences summarized in chapter 3. The rationale behind this approach is to give a detailed analysis of possible fault models (implemented using mutations and failure mode functions), because the available industrial experience cannot be considered as sufficiently reliable resource for deliberately selecting fault models (and potentially dismissing relevant ones). It is expected that the TCG campaigns carried out in the third year of the project will provide more insight into three important aspects and relationships:

a) how software fault models for test models relate to faults in target programming languages,

b) how hardware fault models for test models relate to faults in embedded systems, and

c) how helpful the individual fault models are for fault injection and mutant generation with respect to the significance of resulting test cases.

The generation of mutants started with consideration of available industrial experiences; for instance, those given in table 3-1. It will be investigated during the last year of the project whether those categories of mutations not considered in the first phases but outlined in this report, for instance for action systems (see section 5.1.4) will provide further benefit.

For complex systems, the thoughts and ideas elaborated in chapter 6 will also be regarded. It should be noted, however, that these principles have been brought into MOGENTES during the course of the project, their application has not been deliberately planned at project begin. Therefore, it is to be investigated how MOGENTES could draw maximum benefit.

8 Abbreviations and Definitions

2oo2	2 out of 2
ASIL	Automotive Safety Integrity Level
CSP	Constraint Satisfaction Problem
FMEA	Failure Mode and Effect Analysis
FMF	Failure Mode Function
FTA	Fault Tree Analysis
HIFI	Hardware-Implemented Fault Injection
HIL	Hardware In the Loop
IFW	Input Firewall
MIFI	Model-Implemented Fault Injection
MOGENTES	Model-based GENeration of Tests for dependable Embedded Systems
OFW	Output Firewall
QF	Quality Factor
SAC	Steering Anti Catch-Up
SMV	Symbolic Model Verifier
SUT	Software/System Under Test
SWIFI	Software-Implemented Fault Injection
TMR	Triple Modular Redundancy
VGR	Variable Gear Ratio

9 References

[Acree, 1979] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. *Mutation analysis*. Technical report, School of Information and Computer Science, Georgia Inst. Of Technology, Atlanta, Ga., Sept. 1979.

[AichernigDelgado, 2006] Bernhard K. Aichernig and Carlo Corrales Delgado*; From Faults Via Test Purposes to Test Cases: On the Fault-Based Testing of Concurrent Systems*, FASE 2006, LNCS 3922, pp. 324–338, 2006.

[Ammann, 1999] Paul Ammann and Paul E. Black. *A Specification-Based Coverage Metric to Evaluate Test Sets*. In HASE '99: The 4th IEEE International Symposium on High-Assurance Systems Engineering, pages 239-248, Washington, DC, USA, 1999. IEEE Computer Society.

[Avizienis, 2004] A. Avizienis, J-C. Laprie, B. Randell, and C. Landwehr; *Basic Concepts and Taxonomy of Dependable and Secure Computing*. In IEEE Trans. on Dependable and Secure Computing, Vol. 1, No. 1, January-March 2004, pp 11-33.

[Baumann, 2004] R.C. Baumann, *Soft errors in commercial integrated circuits.* International Journal of High Speed Electronics and Systems, 2004. **14**(2): pp. 299-309

[Black, 2000] Paul E. Black, Vadim Okun, and Yaacov Yesha. *Mutation Operators for Specifications*. In Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00), Washington, DC, USA, 2000. IEEE Computer Society.

[BME, 2009] B. Polgar, et.al; *Pre-Final Framework Implementation*; MOGENTES Deliverable D2.2b, Doc.Nr. 2-05, Ed. 1.0

[Carreira, 1998] J. Carreira, H. Madeira, and J. G. Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. IEEE Transactions on Software Engineering, 24(2):125-136, 1998.

[Clarke, 1994] E. M. Clarke, O. Grumberg, D. Long, *Model Checking and Abstraction,* in ACM Transactions on Programming Languages and Systems, vol 16, no 5, pp 1512-1542, Sept. 1994

[Bensalem et al, 2000] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, LFM 2000: Fifth NASA Langley Formal Methods Workshop, pages 187–196, Hampton, VA, June 2000. NASA Langley Research Center.

[DeMillo, 1978] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. *Hints on Test Data Selection: Help for the Practicing Programmer.* Computer, 11:34-41, 1978.

[Graf, 1997] S. Graf, H. Saïdi, *Construction of Abstract State Graphs with PVS,* in Computer Aided Verification, Lecture Notes in Computer Science, vol 1254, pp 72-83, Springer, 1997

[Han, 1995] S. Han, K.G. Shin, and H.A. Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems. Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium, pages 204-213, 1995.

[ISAAC, 2007] EU FP6-AEROSPACE project reference 501848. Improvement of safety activities on aeronautical complex systems, 2007.

[King et al, 1991] K. N. King, A. Jefferson Offutt. A Fortran language system for mutation-based software testing. Software-Practice & Experience, 21(7):685-718, 1991

[Karlsson, 1994] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneo. Using heavy-ion radiation to validate fault-handling mechanisms. IEEE Micro,

	14(1):8-23, 1994.
[Kumar, 1997]	Kumar K. Goswami, Ravishankar K. Iyer, and Luke Young. Depend: A simulation-based environment for system level dependability analysis. IEEE Transactions on Computers, 46(1):60-74, 1997.
[Kupferman, 2008]	O. Kupferman, W. Li, S. A. Seshia; *A Theory of Mutations with Applications to Vacuity, Coverage, and Fault Tolerance*; Formal Methods in Computer Aided Design, November 2008
[Laprie, 1985]	J.C. Laprie; *Dependable Computing and Fault Tolerance: Concepts and Terminology;* IEEE Int. Symp. On Fault-Tolerant Computing, 1985
[MaOffuttKwon, 2005]	Yu-Seung Ma, Jeff Offutt and Yong Rae Kwon; *MuJava: an automated class mutation system*, Softw. Test. Verif. Reliab. 2005; 15:97–133
[Madeira, 1994]	H. Madeira, M. Z. Rela, F. Moreira, and J. G. Silva. Rie: A general purpose pin-level fault injector. In Proceedings of the 1st European Dependable Computing Conference, pages 199-216, 1994.
[Martins, 2000]	E. Martins and A. C. A. Rosa. A fault injection approach based on reflective programming. In Proceedings of the 2000 International Conference on Dependable Systems and Networks, pages 407-416, 2000.
[McMillan, 1992]	K.L. McMillan. *The SMV system*. Technical Report CMU-CS-92-131, Carnegie-Mellon University, 1992.
[Pataricza, 2002]	A. Pataricza. From the General Resource Model to a General Fault Modelling Paradigm? In Proc. Workshop on Critical System Development with UML, pp 114-115, 2002.
[Pataricza, 2006]	A. Pataricza. Model-based Dependability Analysis. DSc Thesis. Hungarian Academy of Sciences, 2006.
[Pataricza, 2008]	A. Pataricza. Systematic Generation of Dependability Cases from Functional Models. In G. Tarnai and E. Schnieder (eds.): Formal Methods for Automation and Safety in Railway and Automotive Systems. Proc. Symposium FORMS/FORMAT, October 9-10, Budapest, Hungary, pp 17-24, L'Harmattan, Budapest.
[Vesely, 2002]	W. Vesely, M Stamatelatos, J. Dugan, J. Fragola, J. Minarick, J. Railsback; *Fault Tree handbook with Aerospace Applications*; NASA Office of Safety and mission Assurance, 2002
[Vinter, 2005]	J. Vinter, J. Aidemark, D. Skarin, R. Barbosa, P. Folkesson, and J. Karlsson. An overview of goofi - a generic object-oriented fault injection framework. Technical Report 05-07, Department of Computer Science and Engineering, Chalmers University of Technology, 2005.
[Vinter, 2007]	J. Vinter, L. Bromander, P. Raistrick, and H. Edler. Fiscade - a fault injection tool for scade models. In Proceedings of the 3rd IET Conference on Automotive Electronics, pages 1-9, 2007.
[WeiglhoferAichernig Wotawa, 2009]	Martin Weiglhofer, Bernhard Aichernig, and Franz Wotawa; *Fault-based conformance testing in practice.* International Journal of Software and Informatics, 3(2–3):375–411, June/September 2009.