MOGENTES
Model-based Generation of Tests for Dependable Embedded Systems

Integrated Testing Methods and Techniques

<table>
<thead>
<tr>
<th>Project</th>
<th>MOGENTES</th>
<th>Contract Number</th>
<th>216679</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document Id</td>
<td>4-09_1.1r_D4.4</td>
<td>Date</td>
<td>2009-12-29</td>
</tr>
<tr>
<td>Contact Person</td>
<td>Rickard Svenningsson</td>
<td>Organisation</td>
<td>SP</td>
</tr>
<tr>
<td>Phone</td>
<td>+46 10 516 57 89</td>
<td>E-Mail</td>
<td><a href="mailto:rickard.svenningsson@sp.se">rickard.svenningsson@sp.se</a></td>
</tr>
<tr>
<td>Name</td>
<td>Company</td>
<td>Department</td>
<td>No. of copies</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------</td>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td>All MOGENTES team members</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
## Change History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Reason for Change</th>
<th>Pages Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>2009-10-16</td>
<td>Initial version</td>
<td>All</td>
</tr>
<tr>
<td>0.2</td>
<td>2009-11-30</td>
<td>Draft version</td>
<td>All</td>
</tr>
<tr>
<td>0.3</td>
<td>2009-12-18</td>
<td>Final draft</td>
<td>All</td>
</tr>
<tr>
<td>1.0</td>
<td>2009-12-22</td>
<td>Document release</td>
<td>All</td>
</tr>
<tr>
<td>1.1</td>
<td>2009-12-29</td>
<td>Updated release</td>
<td>All</td>
</tr>
</tbody>
</table>
Contents of Figures

Figure 2-1: Two-tank System .................................................................8
Figure 2-2: Verification Result between 'as' and 'asM' .................................10
Figure 3-1: Model-Driven Architecture (MDA) ...........................................11
Figure 3-2: A Simulink model with a block .............................................12
Figure 3-3: A block-type defined in terms of ANSI-C .................................12
Figure 3-4: Unwinding Simulink models ................................................13
Figure 3-5: A simple Simulink program and its mutation ............................15
Figure 3-6: Mutations and Faults injected into a simple Simulink model ........17
Figure 4-1: Simulink behavior model extended with one failure mode function (Value Stuck-At-Zero) ........22
Figure 4-2: Reduced experiment set after pre-injection analysis ..................23
Figure 4-3: FI time window ....................................................................24
Figure 4-4: FI location for a symmetric fault (the circle on input signal Temperature Sensor) and location for an asymmetric fault (the square) ...........................................................24
Figure 4-5: Detailed description of the temperature conversion subsystem ........25
Figure 4-6: Simulink model of a TMR pedal sensor voter ............................27
Figure 4-7: XML example of a fault injection test case ...............................29
Figure 4-8: Transient fault .....................................................................29
Figure 4-9: SP MODIFI Graphical User Interface; Step 1 .........................30
Figure 4-10: SP MODIFI Graphical User Interface; Step 2 ......................31
Figure 4-11: SP MODIFI Graphical User Interface; Step 3 .......................32
Figure 4-12: SP MODIFI Graphical User Interface; Step 4 .......................33
Figure 4-13: SP MODIFI Graphical User Interface; Step 5 .......................34
Figure 4-14: SP MODIFI Progress Visualization .......................................35
Figure 4-15: SP MODIFI Analysis Tool ..................................................35
Figure 4-16: Graphical representation of the XML schema for the description of FMFs ........................36
Figure 4-17: An example of an XML description of an FMF .......................37
Figure 5-1: Prover iLock usage process ..................................................39
Figure 5-2: Prover iLock track overview ..................................................40
Figure 5-3: Prover iLock variable assignment .........................................41
Figure 5-4: Graphical step visualization ..................................................42
Figure 5-5: Prover iLock verification result .............................................42
Figure 5-6: Graphical step visualization ..................................................43

Contents of Tables

Table 4-1: Applicable FMFs for different Locations ..................................27
Table 4-2: Example FMF ........................................................................36
1 Introduction

This report aggregates the new test case generation methods and techniques that have been developed within the MOGENTES project. The four different techniques for test case generation described in this report are:

- Mutation-based test-case generation for Qualitative Action Systems (QAS) using the Ulysses tool
- Mutation- and fault injection based test-case generation for Simulink models using Bounded Model Checking (BMC)
- Fault injection based test-case generation for Simulink models using the SP MODIFI tool
- Fault injection and model checking based test-case generation for PiSPEC models using the Prover iLock tool

The current version of the Ulysses tool is able to check the conformance between two given QAS. The result, which contains the discriminating behavior between an original and a mutated QAS, can be used to extract abstract test cases by a reachability analysis. QAS can be used to describe hybrid systems where the continuous part is modeled with a technique called Qualitative Reasoning. A system model, in the form of a Qualitative Action System specification, could in a future version of Ulysses be used as a test oracle for test cases generated by the Bounded Model Checking approach.

In mutation-based test-case generation using BMC, test cases are automatically generated from a model by mutating it to another model and disproving the equivalence of the two models. This is done using model checking, which is a technique that can be used for demonstrating, using formal proof, that a system satisfies a set of properties. The generated counterexample, as the result from the model checking, can be used as a test case. The test case generation approach is based on Simulink models which are transformed to action systems (described in the C language) for model checking. BMC also supports fault injection for generation of test cases that can be verified by SP MODIFI.

The SP MODIFI tool performs fault injection on Simulink models. Faults can be injected on any output from a Simulink block including complex blocks such as state machines and compiled code blocks. Failure Mode Functions (FMF), i.e. implementations of effects of hardware faults, are modeled into standard Simulink blocks which are inserted in the system model and activated during simulation. With SP MODIFI it is possible to create a Failure Modes and Effects Analysis (FMEA), but also to generate an optimized set of test cases which represents minimal cut sets (MCS), i.e. the minimal combination of faults that violates a safety requirement.

The Prover iLock-based track for test case generation is based on model checking and is used for test case generation from PiSPEC models of rail yards. Typical properties that are verified using this formal verification technique are safety properties, stating some desired property that the system should always satisfy (alternatively, a safety property expresses unwanted behaviour that the system should never allow). While SP MODIFI uses fault injection to carry out an automated and dynamic FMEA (bottom-up approach) to find MCS, Prover iLock generates MCS based on model checking, which is similar to the work that is done in a traditional Fault Tree Analysis (FTA), which is a top-down approach. SP MODIFI is suitable for large complex models with e.g. floating point arithmetic while Prover iLock is developed for models using arithmetic with booleans. The two approaches used by the tools for MCS generation are thus complementary and can, integrated into one tool-chain, improve model-based safety analyses for a wide range of applications.

The remainder of this deliverable is organized as follows: Chapter 2 describes how test cases can be generated for hybrid systems (i.e. systems with continuous evolution and discrete control) by using Qualitative Action Systems. Chapter 3 describes how test cases can be generated by the usage of mutation testing with BMC. Chapter 4 presents a model-implemented fault injection technique for test case generation and the SP MODIFI tool. Chapter 5 describes how fault injection in combination with model checking is used in Prover iLock for generation of minimal cut sets.
1.1 Partner Contributions
This document has been produced by SP Technical Research Institute of Sweden with contributions from Graz University of Technology (Chapter 2), Swiss Federal Institute of Technology Zürich / University of Oxford (Chapter 3) and Prover Technology AB (Chapter 5).
2 Test Case Generation from Qualitative Action Systems

Many real systems exhibit continuous properties which cannot be modeled at sufficient detail for test case generation with existing discrete approaches. Conceptual examples are the Steering Anti-Catchup (SAC) from FFA or the bucket control of RELAB. Qualitative Action Systems (QAS) offer a way to cope with that challenge.

The current version of the prototype tool Ulysses is able to check the conformance modulo ioconf between two given Qualitative Action Systems (QAS) [Aichernig, 2008]. The result is an Input Output Labeled Transition System (IOLTS) which contains the discriminating behavior between an original and a mutated QAS. From this IOLTS abstract test cases can be extracted by a reachability analysis from the fail states to the initial state. The fail states are not part of the test case as they are implicit, i.e. every unspecified output event causes a transition to a fail state.

QAS can be used to describe hybrid systems where the continuous part is modeled with a predicate abstraction technique called Qualitative Reasoning [Kuipers, 1994]. Here, the infinite state space of continuous functions is converted to finite transition systems. The discrete part of the system consists of conventional actions which may have names (labels) with optional parameters. Named actions are partitioned into input and output actions. The remaining unnamed actions are internal and, during their execution, cause internal events. The state space exploration of the system yields an IOLTS which we use for conformance verification. Ulysses computes the verification result on-the-fly without generating the full IOLTS of both QAS.

In a future version of Ulysses a system model in the form of a QAS can be used as test oracle for white-box test cases generated by the project partner ETH. This will be an application of monitoring where executed white-box test cases are replayed on a QAS specification.

Since the tool is still in an evaluation phase the small two-tank example in Figure 2-1 demonstrates the approach. The following requirements specify the behavior of the system:

If the water level in T2 decreases below a certain Reserve mark and T1 is full, pump P1 starts pumping water until T2 is full or T1 gets empty. In addition, the controller needs to control the pump P2 that is pumping water out of T2: P2 shall be turned on as long as a button WaterRequest is pressed and there is enough water in T2 (T2 not Empty).

The two-tank example system in Figure 2-1 can be described with the following QAS specification:

as :-
out_pump1_on : (qEval(t2 < reserve) #\ qEval(t1 = full) #\ p1 #= 0) => (p1 := 1, inout := flow:zero..max/std),
out_pump1_off : (p1 #= 1 #\ (qEval(t1 < empty) #\ qEval(t2 = full))) => (p1 := 0, inout := flow:zero/std),
out_pump2_on : (wr #= 1 #\ p2 #= 0 #\ qEval(t2 > empty)) => (p2 := 1, out := flow:zero..max/std),
out_pump2_off : (p2 #= 1 #\ (wr #= 0 #\ qEval(t2 <= empty))) => (p2 := 0, out := flow:zero/std),
in_water_req([X]) : ((wr #= 0 #\ qEval(t2 = full) #\ X #= 1) #\ (wr #= 1 #\ qEval(t2 < reserve) #\ X #= 0)) => (wr := X),

\([t1(X1), t2(X2)] : (#\ ((qEval(t2 < reserve) #\ qEval(t1 = full) #\ p1 #= 0) #\ (p1 #= 1 #\ qEval(t1 < empty) #\ qEval(t2 = full))) #\ (wr #= 1 #\ p2 #= 0 #\ qEval(t2 > empty)) #\ (p2 #= 1 #\ (wr #= 0 #\ qEval(t2 <= empty))) #\ (wr #= 0 #\ qEval(t2 = full)) #\ (wr #= 1 #\ qEval(t2 < reserve))))

\(~> (In := (in),
Out := (out),
Inout := (inout),
In := D1 + Inout,
Inout := D2 + Out,
D1 := dt X1,
D2 := dt X2).\n
Discrete actions in the system may have a label (name) with possible arguments. Unnamed actions are internal. In the example ‘as’ is the name of system model and the first five actions are named discrete actions. Each action has a guard followed by ‘=>’ and then the action body. When the guard of an action is satisfied the statements in the body are executed. The state variables ‘p1’ and ‘p2’ are of type bool whereas the remaining variables have a qualitative type. A detailed description of the example can be found in [Aichernig, 2008]. The action ‘in_water_req([X])’ models the user interaction with the system. A user turns on the water when T2 is full and turns it off when the level in T2 drops below ‘Reserve’. The on/off behavior of this action is reflected in the Boolean parameter ‘X’.

The last action is called a qualitative action ‘A : B ~> C’ where A specifies the set of observable variables, B is the evolution guard which defines the domain of the qualitative differential equations in C. The evolution of a qualitative action terminates in states where the evolution guard is violated. Qualitative actions nondeterministically update the qualitative state variables according to the initial state and the QDEs in C. The execution of qualitative actions results in a ‘qual’ event, which contains the valuation of qualitative variables at the end of the action. Hence, ‘qual’ is considered as output event (action).

Depending on the execution of qualitative actions different discrete actions get enabled over time. Thus, the specification describes both parts, the environment and the controller forming a hybrid system. The exploration of the QAS with name ‘as’ from a given initial state yields sequences of event traces resulting in an IOLTS. Based on the LTS semantics we apply mutation-based test case generation. For this purpose a specification and a mutated version of the specification in the form of two LTSs are compared regarding a conformance relation, in our case ioconf. The result is an LTS over system inputs and outputs which contain two kinds of final states: ‘pass’ and ‘fail’. When the mutant allows an unspecified input event in a state where the original specification does not allow it, this event leads to a pass state. Behavior which would follow after such ‘pass’ states is not considered by the conformance relation as only traces of the specification are of relevance.

However, when an output action is executed in a state inside the specification where it is not allowed to, a ‘fail’ state follows this output action. Note, that the non-determinism after ‘qual’ actions is resolved when it comes to the extraction of test cases from the product graph.

In order to illustrate the approach we introduce a mutation into the following action

‘out_pump2_off : (p2 #= 1 #\ (wr #= 0 #\ qEval(t2 <= empty))) => (p2 := 1, out := flow:zero/std)’. The mutated specification is denoted as ‘asM’ and the tool Ulysses computes the synchronous product modulo ioconf between ‘as’ and ‘asM’. The result of the product calculation is shown in Figure 2-2. The LTS has one input action (with ‘_in’ prefix) and output actions (with ‘_out’ prefix).

In the initial state of system both tanks are empty, all flows are set to zero except the inflow into tank T1 which is set to some positive value. According to the controller specification, see [Aichernig, 2008], pump P1 is switched on when tank T1 reaches the ‘Full’ level and tank is lower than ‘Full’. Hence, before P1 becomes enabled the level in T1 has to increase from zero to ‘Full’. Then, when pump P1 is running, the water level in
T2 increases which may have two effects: either T1 is full and the user of the system issues a water request or T2 is not yet filled as pump P1 has been turned off because of low water in tank T1. The cycle of switching P1 on and off is repeated until T2 gets full. Then the water request causes pump P2 to be turned on which lowers the fill level in T2. When the level in T2 drops below ‘Reserve’ the water request is stopped and the controller sends the command to turn off pump P2. However, in a faulty implementation pump P2 keeps running after the turn off command. This misbehavior is detected by the conformance check when the controller tries to issue a second ‘out_pump2_off’ command. This leads to a fail state. A correct implementation shows a change in the continuous behavior of the system, i.e. a ‘qual’ event, since a flow has changed, i.e. pump P2 does not pump anymore.

Figure 2-2: Verification Result between ‘as’ and ‘asM’

This short example gives the idea behind mutation-based test case generation for hybrid systems. A more detailed description of the background theory can be found in [Aichernig, 2008].
3 Mutation Based Test Case Generation

The following sections continue and detail the approach to mutation-based test-case generation (TCG) using (bounded) model checking that is given in deliverable D4.1 “Fault-based Test-Case Generation Methods.” Following the method in D4.1, test-cases are automatically generated from a model or implementation \( M \) by mutating \( M \) to \( M' \) and disproving the equivalence of \( M \) and \( M' \) by means of model checking (w.r.t. a suitable notion equivalence, see D4.1). If \( M \) and \( M' \) are not equivalent, a model checker will be able to generate a counterexample demonstrating this fact: the model checker will find input values such that the execution of \( M \) and \( M' \) produces different outputs.

Going beyond D4.1, the next pages refine the mutation-based TCG approach. For models of realistic complexity, the number of mutations that can be applied grows rapidly in the size of the model, in the number of mutation types that are considered, and in the number of mutations to be injected simultaneously. This entails that a naive approach to mutation-based TCG, invoking a model checker once for each mutation, has a prohibitive runtime. We show how to overcome this problem, namely (i) how to efficiently generate small test-suites that kill large populations of mutants, exploiting the locality of mutations and redundancy of test-cases, and (ii) how to efficiently filter out those mutations that do not have any effect on the output of a system, i.e., mutations such that \( M \) and \( M' \) are equivalent.

3.1 Overview of TCG using Model Checking and Mutations

3.1.1 Model-Driven Architecture

The MOGENTES test-case generation approach is based on the Model-Driven Architecture (MDA). Platform-independent models (PIMs) given in a high-level modelling language are automatically transformed in to platform-specific models (PSMs) (see Figure 3-1).

![Figure 3-1: Model-Driven Architecture (MDA)](image)

One of the supported modelling languages in the MOGENTES project is Simulink. Simulink models are synchronous data-flow models. We use action systems (represented in terms of the ANSI-C language) to represent the platform-specific models. We provide a description of the transformation process in the next section.

3.1.2 Simulink Models

Due to the lack of a formal semantics, Simulink models are not directly amenable for an automated analysis. Therefore, we chose to translate Simulink into an intermediate language. The MDA approach allows us to separate the language semantics from the formal analysis. We generate an intermediate model represented in terms of action systems (stored as ANSI-C programs).

Simulink models (such as the one depicted in Figure 3-2) contain signals and blocks. In our approach, block-types are defined as ANSI-C libraries. These (hand-crafted) ANSI-C libraries are made available as source code and provide the formal semantics for the various block-types of Simulink.

A block-type is defined by

- **three structs**:
  - \( \text{inp}_t \): defines the input signals
  - \( \text{out}_t \): defines the output signals
  - \( \text{prop}_t \): provides block-specific properties
- and one function: \( \text{out}_t \text{ semfun}(\text{prop}_t* p, \text{inp}_t* \text{in}) \).
typedef struct { float i1; float i2; } Foo_inp_t;
typedef struct { float o; } Foo_out_t;
typedef struct { float UpperLimit, LowerLimit; } Foo_prop_t;

Foo_out_t Foo_semfun (Foo_prop_t *p, Foo_inp_t *in) {
    ...
}

Figure 3-3: A block-type defined in terms of ANSI-C

Figure 3-3 shows the respective data-structures for the example in Figure 3-2. The semantics of the foo operation is defined by the body of the function Foo_semfun, which is omitted here. Thus, it is easy to add additional block-types to our transformation tool.

Status of the Simulink Front-End. Our Simulink frontend currently supports discrete dataflow, floating-point arithmetic, subsystems, and feedback loops. Table 1 shows the block-types currently supported by our transformation tool. We are currently working on supporting Stateflow diagrams, S-functions, and additional block-types such as integrators, derivatives, discrete transfer functions, switches, trigonometric blocks, delay blocks, and signal conversion blocks.

The analysis and test-case generation approach for the resulting ANSI-C models is explained in detail in the sections below.

3.1.3 Model Checking

Our test-case generation approach for Simulink models is based on a verification technique known as “Model Checking.” Model checking, in the most general sense, is a technique that explores the reachable states of a model in order to determine whether a given specification is satisfied [Clarke et al., 1999]. It differs from testing in so far as it aims at covering the entire state space of the model or program under test, providing a correctness guarantee that is rarely achieved by means of testing. If the specification is violated, Model Checking tools are often able to provide a counterexample, i.e., a witness for the incorrectness of the model.

Bounded Model Checking (BMC) is a variation of the model checking technique which only explores execution traces up to a certain (user-defined) length $k$. This technique either provides a guarantee that the first $k$ execution steps of the program are correct (with respect to the property to be verified) or a counterexample of length at most $k$. The ability to report counterexamples is the essential feature we use to generate test cases. While Model Checking is able to prove the correctness of software or systems (in contrast to testing), it practically has certain disadvantages: scalability is usually worse than that of testing, it can be difficult to model physically complex environments, and software model checking (on the level of source code or models) is performed under the assumption of correct compilers and hardware.

Figure 3-4a shows the schema of a Simulink model with a feedback loop. A bounded model checking algorithm unwinds such models as indicated in Figure 3-4b; the signals $i_i$ and $o_i$ refer to the input and output signals in the $i^{th}$ step (or point in time), respectively.

<table>
<thead>
<tr>
<th>BusCreator</th>
<th>Gain</th>
<th>Rounding</th>
</tr>
</thead>
<tbody>
<tr>
<td>BusSelector</td>
<td>Logic</td>
<td>MinMax</td>
</tr>
<tr>
<td>Clock</td>
<td>Math</td>
<td>Saturate</td>
</tr>
<tr>
<td>Constant</td>
<td>Memory</td>
<td>Sum</td>
</tr>
<tr>
<td>DatatypeConversion</td>
<td>Product</td>
<td></td>
</tr>
<tr>
<td>Display</td>
<td>RelationalOperator</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Handled block-types
For the purpose of test-case generation it suffices to determine whether certain states in a model are reachable. A model is specified by a formula representing a (possibly partial) transition relation $R$ (e.g., specified by means of an action system or a C program) and a predicate $I$ that determines the valid initial states of the model. The transition relation $R$ relates the current state of the model to its successor states (i.e., the potential states after one step). The structure of $R$ may be further detailed by means of a control flow graph, which partitions $R$ into a separate transition function for each program location. This simple formalism is sufficiently general to allow imperative models (such as C programs or state charts) as well as data flow models (such as Simulink models). Furthermore, the predicate $I$ characterises the set of valid initial states of the model (this may be the safe or reset state of the system), i.e., $I(s)$ holds if $s$ is a valid initial state.

A path (or execution trace) $\pi$ of the model is a sequence of states $s_0, s_1, \ldots, s_n$ such that the adjacent pairs $s_i, s_{i+1}$ of states in that sequence are related by $R$ (i.e., $\bigwedge_{i=0}^{n-1} R(s_i, s_{i+1})$), and $I(s_0)$ holds (i.e., $s_0$ is a valid initial state). A state is induced by the values of the variables (or wires) of the model. In reactive models, the variables are typically partitioned into input variables, hidden (or internal) variables, and output variables. The observable part of an execution trace is therefore the sequence of inputs and the resulting sequence of outputs. Given a state $s_i$, we use $s_i.i$ to refer to the input, and $s_i.o$ to denote the output.

### 3.1.4 Equivalence Checking

(Formal) Equivalence Checking is a technique used to formally prove that two models $M$ and $M'$ exhibit the same observable behaviour [Kuehlmann and van Eijk, 2002]. This is achieved by comparing the input and the output behaviour of the two models. To construct a test-scenario (i.e., a sequence of input/output pairs), we are interested in checking whether for a given input sequence the outputs in the first $k$ steps of the executions of the models match. This notion of "$k$-equivalence" is decidable, assuming that the input and output values have a finite range.

Whether two given models are $k$-equivalent can be decided using model checking. Given two models $M$ and $M'$ (comprising the transition functions $R$ and $R'$ and the initial state predicates $I$ and $I'$, respectively), we can check (assuming states of finite size) whether

\[
\begin{align*}
\bigwedge_{i=0}^{k-1} s_i.i &= s'_i.i \land I(s_0) \land \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \land I'(s_0') \land \bigwedge_{i=0}^{k-1} R'(s'_i, s'_{i+1}) \land \bigwedge_{i=0}^{k} s_i.o \neq s'_i.o
\end{align*}
\]

(1)

is satisfiable. Any satisfying assignment to this formula represents two executions of $M$ and $M'$ that yield a different output sequence. The models are equivalent if Formula (1) is unsatisfiable. Checking software equivalence is more complicated, since the output of the models may occur at different times. The algorithmic details of bounded model checking-based equivalence checking are covered in [Kroening et al., 2003]. A predicate-abstraction [Graf and Saïdi, 1997] based approach is presented in [Kroening and Clarke, 2004].
The test case generation techniques described above enable the generation of test suites that satisfy structural coverage criteria such as condition or statement coverage. The coverage criteria for fault-based testing in the MOGENTES project are based on syntactic and semantic modifications of the model. Given a modification to the model, the aim is to generate a test case that demonstrates the resulting change of the behaviour. Simple structural coverage metrics are not sufficient, since even strong coverage criteria such as MC/DC provide no guarantee that the error resulting from the modification of the model has a visible impact on the behaviour generated by exercising the test suite. In contrast, checking the equivalence of a model $M$ and its modified counterpart $M'$, allows us to find input sequences $s_0, i, \ldots, s_k, i$ for which the models yield (sufficiently) different output sequences.

The fault driven test case generation approach is inspired by mutation testing and fault injection:

- **Mutation testing** denotes the method of making (syntactic) modifications to the source code of the implementation. The intention is to evaluate a given test suite based on whether it is able to detect the introduced faults and to aid the generation of additional meaningful test cases.

- **Fault injection.** Fault injection triggers the occurrences of faults in the system under test. The main purpose of this technique is to evaluate the error handling mechanisms of the system.

The common idea underlying both approaches is to make modifications to the system and to run test cases that demonstrate the impact of these changes. Our aim is to handle software mutations and fault injection in a uniform manner: The model-based approach of the MOGENTES project allows us to embed software mutations as well as hardware faults in the model of the system by using a set of fault models. A catalogue of mutations and fault models is provided in Deliverable D3.1b. In this section, we focus on describing an automated analysis that allows us to extract “good” test cases from a model containing mutations and failure modes.

### 3.1.5 Mutation Testing and Fault Injection

From an abstract point of view, mutations as well as injected faults are simply modifications to the behaviour of the model.

**Example 1** Consider the simple Simulink diagram in Figure 3-5a. The input signals $i_1$ and $i_2$ are related to the output signal $o$ by means of the formula $o = i_1 \times i_2$, i.e., the transition function is

$$R(s_i, s_{i+1}) \overset{def}{=} s_i.o = s_i.i_1 \times s_i.i_2 \land s_{i+1}.o = s_{i+1}.i_1 \times s_{i+1}.i_2.$$ 

A possible syntactic mutation is to replace the multiplication ($\times$) with an addition:

$$R'(s_i, s_{i+1}) \overset{def}{=} s_i.o = s_i.i_1 + s_i.i_2 \land s_{i+1}.o = s_{i+1}.i_1 + s_{i+1}.i_2.$$ 

This mutation can be implemented in the diagram using an enable signal, allowing us to switch the mutation on and off (see Figure 3-5b).

In our formalism, such a modification can be modelled by replacing the transition relation $R$ of our model $M$ with a slightly modified transition relation $R'$ (and possibly a modified condition $I'$ for the initial state). The faults introduced into the system may be either permanent, transient, or intermittent (i.e., repeatedly). The former case can be simply modelled by permanently altering the transition relation $R$, and applying the resulting relation $R'$ in each step:

$$I'(s_0) \land R'(s_0, s_1) \land R'(s_1, s_2) \land R'(s_2, s_3) \land \ldots$$

To model transient or intermittent faults, we have to take the temporal aspect into account, i.e., the alteration becomes only effective at certain points in time. A typical execution satisfies the following constraint:

$$I(s_0) \land R(s_0, s_1) \land R(s_1, s_2) \land R'(s_2, s_3) \land R(s_3, s_4) \land \ldots$$

The kinds of faults and mutations introduced into the model are the subject of the MOGENTES Deliverable D3.1b. Mutations are small syntactic changes of the model, whereas simulated hardware faults require semantic changes to the model that reflect physical faults of the system as accurately as possible. Conceptually,
Figure 3-5: A simple Simulink program and its mutation

however, there is no difference when it comes to their integration into the transition relation: The implementation of faults in the model \( M \) requires syntactic changes to \( M \).

Depending on the extent of these modifications, the resulting error may not be immediately observable, i.e., it is not necessarily the case that

\[
s_{0,i} = s'_{0,i} \land R(s_0, s_1) \land R'(s'_0, s'_1) \implies s_{1,o} \neq s'_{1,o}
\]

holds. Even though \( s_1 \) differs from \( s'_1 \), the outputs \( s_{1,o} \) and \( s'_{1,o} \) may be indistinguishable: the modification of \( R \) may not necessarily have an (immediate) impact on the observable behaviour. Intuitively, a test case is “good” if it yields a different outcome for \( M \) and \( M' \). In mutation testing, the term weak mutation testing refers to the condition that the test cases should cause different program states for the mutant and the original model. In the case that the affected part of the state is not observable, this condition is not sufficient for our purpose. Strong mutation testing refers to the case where the error propagates to the output of the model and is caught by an appropriate test case. In dependable systems, this notion may be too strong, since redundant systems may tolerate a certain number of faults. Note that this case can be detected using a complete Model Checking technique or \( k \)-induction (for details we refer the reader to our survey [D’Silva et al., 2008]).

3.1.6 Generating Test Cases

As mentioned previously, one way of generating test cases that detect a mutation is to find a satisfying assignment to Formula (1). Such a satisfying assignment provides the inputs that yield a different output sequence during the first \( k \) steps and, provided the observable behaviours of the two models \( M \) and \( M' \) are not fully equivalent, such a solution must exist for some \( k \).

Encoding Combinations of Faults  Assume that the objective is to generate a test suite that detects single faults (or mutations). The naïve approach to create such a test suite is to generate a new model \( M' \) for each conceivable fault or mutation and to generate an instance of Formula (1) for each pair of models \( M \) and \( M' \). In practise, this approach is very wasteful, since modern satisfiability checkers such as MiniSAT [Eén and Sörensson, 2004] are able to solve problem instances incrementally. Encoding \( M' \) in a way such that faults or mutations can be activated or deactivated by adding constraints to the formula allows the SAT solver to (partially) reuse the information it has already derived.

Therefore, we propose to generate a modified model \( M' \) that contains all faults and mutations for which we want to generate test cases. We use the same idea as in Figure 3-5b and introduce a Boolean flag \( f \) for each modification that allows us to activate (deactivate) the respective fault/mutation by setting \( f \) to true.
(false). Assume that $R$ is the transition relation of the original model $M$ and that $R_i'$ is the transition relation of the model $M_i'$ that contains the $i^{th}$ of the $n$ modifications in question. We define the model $R_\mu$ as follows:

$$
R_\mu(s_0, s_1, f_1, \ldots, f_n) :=
\begin{cases}
R(s_0, s_1) & \text{if } \bigvee_{i=1}^n f_i = 0 \\
R_1'(s_0, s_1) & \text{if } f_1 = T \land \bigvee_{i=2}^n f_i = 0 \\
R_2'(s_0, s_1) & \text{if } f_2 = T \land f_1 = 0 \land \bigvee_{i=3}^n f_i = 0 \\
\vdots
\end{cases}
$$

(2)

We use $M_\mu$ to denote the model with the transition relation $R_\mu$.

Given the resulting transition relation $R_\mu$ (as defined in (2)) we can construct an instance of Formula (1). A fault in the modified model $M_\mu$ can be triggered by adding a constraint of the form

$$
F_j := f_j \land \bigwedge_{0<i\leq n, i\neq j} \neg f_i .
$$

(3)

**Example 2** Figure 3-6 shows a mutation and a fault injected into the Simulink diagram in Figure 3-5a. The first diagram shows the implementation of a signal-stuck-at-0 fault. The diagram below combines this fault and the mutation in Figure 3-5b into one model. The model provides two flags allowing us to trigger the mutations.

The decision procedure on which the CBMC tool is based on performs bit-level accurate reasoning by transforming the instance of Formula (1) into an equi-satisfiable propositional formula $EQ_k$ in Conjunctive Normal Form $^1$ (CNF). This formula is then handed over to the satisfiability checker MiniSAT [Eén and Sörensson, 2004]. The decision process of MiniSAT is incremental, i.e., it allows

- to add additional clauses, and
- to add or remove a constraint $F_j$ of the form described in (3)

without restarting the solver, meaning that the solver can reuse intermediate results if it has to solve similar problem instances.

Let $EQ_k$ denote the CNF of the instance of Equation (1) derived from the models $M$ and $M_\mu$. We can generate a test suite covering all $n$ mutations in question by iteratively computing satisfiable assignments to

$$
EQ_k \land F_i, \quad i \in \{1, \ldots, n\}.
$$

(4)

If each of the instances of Formula (4) has a solution, we obtain $n$ (not necessarily different) test cases that cover all mutations injected into the model $M_\mu$.

### 3.2 TCG for Large Numbers of Mutations

#### 3.2.1 Finding an Efficient and Sufficient Test-Suite

The technique described above uses a SAT-solver to extract $n$ test cases from Formula (4), each of which corresponds to one mutation. If the number of mutations ($n$) is large, this may lead to an equally large number of test cases. These test-cases are computationally expensive to generate and time-consuming to execute. Therefore, it is desirable to minimise the size of the test-suite.

**Example 3** Consider the Simulink diagrams in Figures 3-5a, 3-5b, and 3-6. Assume that we use the model in Figure 3-6b to generate a test case and assume that both mutations are enabled. Consider the test case $s.i_1 = 0.0$ and $s.i_2 = 23.4$:

<table>
<thead>
<tr>
<th>Input signal 1</th>
<th>Input signal 2</th>
<th>Fault (3)</th>
<th>Mutation (4)</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>23.4</td>
<td>off</td>
<td>off</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>23.4</td>
<td>on</td>
<td>off</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>23.4</td>
<td>off</td>
<td>on</td>
<td>23.4</td>
</tr>
<tr>
<td>0.0</td>
<td>23.4</td>
<td>on</td>
<td>on</td>
<td>23.4</td>
</tr>
</tbody>
</table>

$^1$A propositional formula in conjunctive normal form is a conjunction of disjunctions of literals, where a literal $i$ is a propositional variable or its negation (e.g., $a$ or $\neg a$). A clause is a disjunction of literals.
(a) Fault Injection: A signal-stuck-at-0 fault

(b) The fault and the mutation from Figure 3-5b combined in one model

Figure 3-6: Mutations and Faults injected into a simple Simulink model
This test case is sufficient to detect the combination of the syntactic mutation and the injected signal-stuck-at-0 fault, as well as the single syntactic mutation. However, it fails to detect the signal-stuck-at-0 fault.

Example 3 shows that it is in general not possible to derive a test-case that covers more than one mutation by simply activating several mutations simultaneously in a single instance of Formula (4). It is, however, possible that a test-case \( t \) derived from a model \( M_\mu \) with a single mutation detects other mutations \( \nu \), too. This can be efficiently checked by evaluating the behaviour of the mutated model \( M_\nu \), for the input defined by the test-vector \( t \). The execution of a given test-case on a model is very efficient compared to the model checking-based computation of a new test-vector for \( M_\nu \).

Example 4 We continue working in the setting of Example 3. Assume that we use the model in Figures 3-5a and 3-5b to generate the test case \( s.i_1 = 1.0 \) and \( s.i_2 = 23.4 \). We can compute the outcome for all different models for this input by simply executing the test cases:

<table>
<thead>
<tr>
<th>Input signal 1</th>
<th>Input signal 2</th>
<th>Fault (3)</th>
<th>Mutation (4)</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>23.4</td>
<td>off</td>
<td>off</td>
<td>23.4</td>
</tr>
<tr>
<td>1.0</td>
<td>23.4</td>
<td>on</td>
<td>off</td>
<td>0.0</td>
</tr>
<tr>
<td>1.0</td>
<td>23.4</td>
<td>off</td>
<td>on</td>
<td>24.4</td>
</tr>
<tr>
<td>1.0</td>
<td>23.4</td>
<td>on</td>
<td>on</td>
<td>23.4</td>
</tr>
</tbody>
</table>

This test case is sufficient to detect the syntactic mutation as well as injected signal-stuck-at-0 fault. It fails to detect the combination of both. This is acceptable if we assume a single-fault hypothesis. Therefore, it is not necessary to compute an additional test case.

Note that this test-case cannot be derived from the model in Figure 3-6b if both mutations are active.

In the worst case, \( n \) test-cases are required to cover all \( n \) mutations. In this case, we have to perform \( O(n^2) \) executions of the mutated models and \( n \) model checking runs (one for each of the \( n \) mutations).

To determine the mutants that are killed by a test-case, we intend to use the Modifi tool. This tool already implements a number of optimisations to reduce the required number of system runs.

We are currently considering several additional approaches to increase the efficiency of our test-case generation approach. The following ideas are ongoing work or future work:

- We intend to define metrics on the space of possible mutations to assess the similarity of mutations (this idea is related to the strength of mutations, discussed in Chapter 3.4.1). "Similar" mutations are likely to be covered by the same test-cases and should be checked first, while mutations that are very different can be ignored right away.

- Given metrics of the mutation space, we can apply clustering algorithms to decompose the mutation space. Following the hypothesis that mutations in distinct clusters are unlikely to be covered by the same test-cases, the TCG procedure can be applied independently to each cluster of mutations. This also enables parallelisation of the test-case generation and therefore improves scalability (also see Hussain, 2008).

- Given that our analysis works on a ANSI-C representation of the Simulink model, our model is immediately amenable to "concolic" testing [Godefroid et al., 2005], a successful approach to combine (concrete) test-vectors with symbolic execution. Starting with an execution trace generated by executing a concrete test-case, the corresponding path formula is generated. Additional test cases are generated by changing sub-expressions in this path that correspond to decisions along the path.

- We intend to approximate parts of a model whose behaviour is not observable in the system outputs. Because this is done for a particular test-case/system run, comparatively expensive methods like data-flow analysis are not necessary. Mutations of non-observable parts cannot be covered by the test-case at hand and can be ignored right away. A more detailed discussion of non-observable mutations is provided in Chapter 3.4.

- The size of a given test-suite can possibly be optimised by checking whether certain test-cases are subsumed by other test-cases in the suite. While this does not decrease the computation burden during the test-case generation, it increases the efficiency of the execution of the test-suite.
3.3 Generating Test Cases for Continuous Models

Another issue is the presence of floating-point computations in the models $M$ and $M'$. Currently, no Model Checker is able to handle such computations accurately, because floating-point operations are usually modelled using arithmetic over the reals. This approximation does not take the limited accuracy of floating point numbers into account. In particular, using such an approach, modelling the bit-level faults required by the MOGENTES demonstrators (a catalogue is provided in Deliverable D3.1b) is extremely hard.

A Model Checker supporting floating-point operations is, however, highly desirable since floating-point operations often do not behave as programmers expect. For example, associativity does not hold for floating-point numbers into account. In particular, using such an approach, modelling the bit-level faults required by the MOGENTES demonstrators (a catalogue is provided in Deliverable D3.1b) is extremely hard.

A Model Checker supporting floating-point operations is, however, highly desirable since floating-point operations often do not behave as programmers expect. For example, associativity does not hold for floating-point operations, i.e., $(a \oplus b) \oplus c$ may not be equal to $a \oplus (b \oplus c)$, where $\oplus$ is floating-point addition. Test cases generated by such a Model Checker can thus reveal unexpected results. We extended the Model Checker CBMC to generate test cases for models that have floating-point numbers.

**A New Notion of Equivalence** In the context of fault-based test-case generation, we are also exploring the possibility of relaxing the notion of equivalence. More precisely, in equation 1, if $s_i.o$ and $s'_i.o$ are floating-point outputs, we will replace the constraint

$$\bigvee_{i=0}^{k} s_i.o \neq s'_i.o$$

in equation 1 by the constraint

$$\bigvee_{i=0}^{k} |s_i.o - s'_i.o| \geq \delta_i.$$

That is, instead of requiring a test case to result in a different output, we require the output to violate a tolerance bound given by $\delta_i$. This is reasonable if small deviations in the output are acceptable. Such a relaxation of the notion of equivalence could also speed up the generation of test cases.

Since many fault models listed in Deliverable D3.1b are based on bit-level modifications (such as single-bit-stuck-at faults), there is a strong case for the SAT-based, bit-level accurate decision procedures we apply. In a propositional encoding, single-bit modifications can be integrated very easily, and SAT-solvers can deal with the resulting encoding very efficiently. For instance, modelling a single-bit-stuck-at-1 fault corresponds to setting a single propositional variable in the formula $EQ_k$ to true. Modern SAT-solving algorithms deal with this case by using unit propagation, which is extremely efficient. If, on the other hand, we would use a decision procedure based on the popular Simplex algorithm (an algorithm for solving linear arithmetic equations), encoding the same fault would be complicated and would result in an equation that is very hard to solve.

3.4 Detecting Non-Observability of Mutations

In traditional mutation-based testing, the difficulty to identify mutations without observable effect on the system outputs is known to be one of the main obstacles. Assume that one instance $EQ_k \land F_c$ of (4) (with $i = c$) is unsatisfiable. This indicates that the injected fault corresponding to $f_c$ (see Formula (2)) does not result in an error that propagates to an observable output within $k$ steps. There are two possible reasons for this phenomenon:

(i) The bound $k$ is not sufficiently large to reveal the error.

(ii) The model contains redundancy and the injected fault does not result in an observable change of its behaviour. We say that the model tolerates the fault.

A complete Model Checking algorithm can distinguish both cases. The first case can be addressed by simply increasing the bound $k$. In the second case, the mutation is not strong enough to have any impact on the observable behaviour of the model and the model checking tool provides a proof for the equivalence of the mutated and original model. Depending on the techniques the model checking tool is using, different kinds of proofs can be provided. We aim at investigating whether the presented proof can be reused for other mutations. This is applicable to various notions of a proof:

- Inductive invariants that are computed using abstraction or interpolation techniques (for a new mutation, it is only necessary to check inductivity). We presented two different interpolation techniques in [D'Silva et al., 2010; Kroening and Weissenbacher, 2010].

- Equivalence shown using $k$-induction and a bounded model checker with proof-producing SMT back-end (for a new mutation, it has to be checked whether the SMT proof can be adapted).

- Bounded equivalence that is shown using a bounded model checker with proof-producing SMT back-end.
We expect this approach to be successful whenever the generated proof is “local” in the sense that the component containing the mutation is not responsible for the unobservability (e.g., if dead code has been mutated).

Checking whether a proof can be reused is computationally less expensive than generating a new proof. For instance, checking the inductivity of an invariant can be achieved by bounded model checking, which is more efficient than fixed point-based model checking techniques.

From an equivalence proof, information can be extracted which parts of a system (and its mutant) are relevant for the equivalence. Mutations in irrelevant parts of the system cannot have any influence on equivalence and are covered by the proof right away.

The same techniques as in Sect. 3.2 are applicable: metrics can be formulated to assess the similarity of mutations, and the mutation space can be clustered.

### 3.4.1 Searching for Combinations of Faults

In the case that a mutation is not observable, we can force the occurrence of an error by injecting a stronger fault (as suggested in [Kupferman et al., 2008]). This can be achieved by combining two or more of the faults in the model $M_\mu$. Our setting provides an easy means to trigger more than one fault. Given a set of fault indices $F$ that correspond to the selected set of faults, we construct the constraint

$$F_F := \bigwedge_{i \in F} f_i \land \bigwedge_{i \in \{1, \ldots, n\} \setminus F} \neg f_i.$$  \hfill (5)

By solving the instance $EQ_k \land F_F$, we obtain a test case $TC_F$ that distinguishes the model $M$ and the respective modified model into which the faults/mutations corresponding to $F$ have been injected. While $TC_F$ yields the same output for $M$ and any $M'_\nu$ that contains only a single fault, it is a “good” test case in the sense that it challenges the error handling mechanisms of the implementation.

Our approach to address the cases i and ii described above suffers from two problems:

- Cases i and ii are only distinguishable by means of an (expensive) complete Model Checking procedure.
- In order to avoid a combinatorial explosion of fault combinations, the set $F$ has to be chosen cautiously (also see deliverable D3.1b for industrial experiences with various fault models).

Both issues can only be tackled using heuristics. In our setting, we do not necessarily require a complete verification algorithm, as we are satisfied with a partial result. In order to avoid scalability problems of complete Model Checking procedures, we plan to apply a complete Model Checker to over-approximations of the model, e.g., to localisation reductions $M_\mu$ of $M_\nu$. The reduced model $M_\mu$ is typically orders of magnitudes smaller than $M_\nu$, which enables the Model Checker to pass. This is a conservative procedure for any reachability property, which includes our coverage criteria. If the Model Checker determines that $M_\mu$ tolerates a given fault, so does $M_\nu$, and we do not need to attempt to generate a covering test case. If not so, the Model Checker provides a counterexample on $M_\mu$, which we may attempt to concretize to be satisfied by $M_\nu$ (also see deliverable D3.1b for more information on model abstraction).

The second problem (choosing an appropriate combination of faults or a stronger fault) can be addressed by defining a partial order over combinations of faults that orders them with respect to their strength (as suggested in [Kupferman et al., 2008]). Let $\mu$ and $\nu$ be mutations or faults. According to Definition 1 in [Kupferman et al., 2008], $\mu$ is at least as aggressive than $\nu$ if for every model $M$ and every specification $S$, we have that if $M_\mu$ adheres to $S$, then so does $M_\nu$. For instance, let $\mu$ be a fault that produces the effect that a certain binary signal changes non-deterministically. This fault is stronger than a fault $\nu$ that results in the same signal being permanently 1, since the possible behaviours induced by $\mu$ also contain the behaviour resulting from $\nu$. Note furthermore that a combination of two arbitrary faults $\mu$ and $\nu$ is not necessarily stronger than the single fault $\mu$, since the fault effects might cancel each other out. The aggressiveness ordering introduced above results in a lattice where the bottom element is the empty set of mutations and the top element represents the most aggressive mutation (which is possibly a combination of several mutations). The aim is now to find a combined mutation in the lattice that is strong enough to yield a different output. Since the size of the lattice grows exponentially with the size of $F$, a good heuristic must be chosen in order to avoid combinatorial explosion. Searching the lattice by beginning from the extremes (top or bottom) of the lattice is certainly not a good strategy for the following reasons:

- The closer a combined mutation is to the top element of the element of the lattice, the less likely it is to occur in reality since this would mean that all the mutations occur simultaneously.
• The closer a combined mutation is to the bottom element of the lattice, the less likely it will have an impact on the output since the (combined) mutation might be to weak and might not propagate to the output.

This suggests a binary search on the lattice as proposed by Purandare et al. [Purandare et al., 2009]. That is, the search starts from the middle of the lattice. In case a combined mutation is found not to propagate to the output, all weaker mutations can be deleted from the lattice, since a weaker mutation will not affect the output either. In such a case the search tries to find a stronger mutation. Otherwise, the algorithm outputs the current combined mutation or tries to find a weaker one. Purandare et al. [Purandare et al., 2009] apply a similar technique for checking whether specifications vacuously hold. This work is, as observed by Kupferman et al. [Kupferman et al., 2008], closely related to our problem; the problem of finding mutations that violate a specification is dual to the problem of vacuity checking.
4 Fault Injection Based Test Case Generation

This Section gives an overview of fault injection in general and in particular model-implemented fault injection (MIFI). The problem of minimizing the fault space is also discussed as well as how to generate minimal cut sets which can be used to create efficient test cases for physical fault injection on the final system. Test case generation with the SP MODIFI tool which implements the discussed algorithm is also described. The fault injection technique is similar to the fault injection technique described in the previous chapter, but instead of being used with model checking, it is performed during simulation of a Simulink behavior model. The model-implemented fault injection technique that is focused on in this chapter aims at investigating the robustness against hardware faults.

4.1 Fault Injection

Fault injection [Iyer, 1995] (also known as fault insertion testing) accelerates the occurrences of faults in a system and the main purpose is to evaluate and debug error detection and error recovery mechanisms. It is used at various abstraction levels and phases of the development process. Fault injection is e.g. mandatory in the safety standard IEC 61508 (adapted by the automotive industry as ISO DIS 26262) when the claimed diagnostic coverage is at least 90%. As fault injection has become widely used as an experimental dependability validation method, many different techniques for injecting faults have been developed.

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. Another classification is based on how fault injection mechanisms are implemented. Thus, the focus is on whether extra hardware is 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]. Fault injection mechanisms which are added directly into models of hardware, models of software or models of systems are denoted in MOGENTES as model-implemented fault injection, MIFI [Goswami, 1997], [ISAAC, 2007], [Vinter, 2007].

4.2 Model-Implemented Fault Injection

MIFI is a technique where fault injection mechanisms are inserted into models of either hardware, models of software or models of systems. MIFI can be used for any system that can be modelled unambiguously. In order for a system to qualify for MIFI, it must be described in a suitable modelling language e.g. as Simulink [MATHWORKS, 2009], Scade [SCADE, 2009] or as a PiSPEC [iLock, 2009] specification.

In MIFI the system model is extended into an Extended System Model (ESM) with faults that can be enabled during the experiment in order to evaluate the system behaviour under the influence of these faults. The faults can be triggered individually during the experiment to e.g. evaluate the system behaviour under the influence of multiple faults. Consider the following simple example in Figure 4-1, a Simulink model that has two input signals and one output signal. The output signal value is simply the product of the two input signals. Assume further that the Input signal 1 in Figure 4-1 contains a sensor value. One of several plausible failure modes for sensors is stuck-at-zero (i.e. value is set to zero), which is implemented as a Failure Mode Function (FMF) and encircled in the figure.

Figure 4-1: Simulink behavior model extended with one failure mode function (Value Stuck-At-Zero).
Inputs to the FMF are the nominal signal value as well as *Fault Injection Block Enable*, which is a trigger port for the FMF. This provides a possibility to enable the FMF at any time (and with any duration to simulate e.g. transient and permanent faults) during the simulation. The approach is similar for other model descriptions and can be used both for experimental based evaluation as well as for evaluation by the usage of formal methods (e.g. Model checking) as was done in the ISAAC project [ISAAC, 2007].

### 4.3 Pre-Injection Analysis

Due to the virtually infinite number of faults that are necessary to perform to do an exhaustive analysis of a real system, pre-injection analysis is very important. The purpose of pre-injection analysis is to reduce the fault space into a manageable set of faults that can be executed within a reasonable time without compromising the test case coverage (Figure 4-2). Part of the pre-injection analysis is to remove redundant test cases that are known *a priori* to produce the same result as an already performed test case.

Several techniques have been proposed concerning pre-injection analysis. The Operational-Profile-Based Fault-Injection technique [Guthoff, 1995] only injects faults in live data, i.e. data that will not be overwritten before the next time it is read. Another technique is fault list collapsing [Benso, 1998] which defines a set of rules where various types of faults whose effects can be foreseen from the available fault space are removed. Assembly-level knowledge of the target system can also be used in order to place single bit-flips in registers and memory locations only immediately before these are read by the executed instructions [Barbosa, 2005].

There are two aspects related to pre-injection analysis within the MOGENTES project: minimize the number of faults which has to be injected during development (using MIFI), and minimize the number of faults which has to be injected during system test (using SWIFI or HIFI) by exploring MCSs. Since HIFI experiments are time consuming, especially compared to MIFI experiments, it is important to shrink the set of HIFI experiments necessary to obtain a sufficient coverage as much as possible.

Reduction of the number of MFI experiments can be accomplished by any reduction in the three dimensions: time, location, and FMF. In the following part of this section reduction techniques for each of these three dimensions are presented.

Reduction of the number of FMFs means a reduction of the fault models that are applicable for a specific location in the model. The target applications are often dependable embedded systems where models are used to describe the behavior of a controller software, but also sometimes the behavior of surrounding hardware such as e.g. sensors. Common fault models for embedded systems are EMI or radiation induced transient bit-flips and permanent stuck-at faults originating from manufacturing defects or wear-out, e.g. electro migration. Hence, transient bit-flips and permanent stuck-at (one or zero) fault models can be used...
for those signals (block outputs) which belongs to the microcontroller implemented part of the model. For the rest of the model, if any, different fault models apply for different components. For an analog sensor, offset and gain fault models apply whereas for sensors having a Boolean output, stuck-at fault models apply.

One approach to reduce the time dimension is to limit it by a FI time window, see Figure 4-3. Then faults will only be injected during the FI window time span but the effects can be observed as long as the input stimuli of the workload (model under test) lasts to evaluate the effects of the faults.

![Figure 4-3: FI time window](image)

One technique for reducing the number of fault injection locations in e.g. a Simulink model can be to either only inject fault on block inputs or block outputs. By choosing one or the other the number of locations will be approximately halved since a signal is usually represented by two variables in the generated software, one for the output of a block and another variable for the input of the connecting block. Both options have their advantages and disadvantages. To be able to inject symmetric faults, block outputs have to be the choice (the circle in Figure 4-4). On the other hand, to be able to inject asymmetric fault, the square in Figure 4-4, block inputs has to be the choice.

![Figure 4-4: FI location for a symmetric fault (the circle on input signal Temperature Sensor) and location for an asymmetric fault (the square)](image)

Another way of reducing the number of locations is to inject faults only on e.g. outputs in the top level of the system model. Then when sensitive parts have been identified these parts can be analyzed in more detail. Take the model in Figure 4-4 as an example. Assuming that initial fault injection experiments indicate that
the temperature conversion subsystem block is sensitive, more detailed fault injection experiments targeting the interior of this subsystem, see Figure 4-5, can be run.

![Figure 4-5: Detailed description of the temperature conversion subsystem](image)

**4.4 Fault Injection Based Minimal Cut Sets Generation**

Minimal Cut Sets (MCS) are traditionally computed manually during Fault Tree Analysis (FTA). In FTA, systems are investigated in a top-down manner, starting from a dangerous or undesired event, called top-level-event. The possible causes of this top-level event are then derived through system analysis. The causes found are then considered top-level events and the procedure goes on until the fault tree is complete, i.e. when the causes are considered to be atomic. This is a time-consuming process which can be automated as described in this chapter.

The process of how to generate MCS from models is general with respect to the overall algorithm, hence does not only apply for Simulink.

Starting from the complete set of fault combinations (i.e. combinations of location, time and failure mode function), the model is explored by injecting single faults (i.e. single fault combinations). If a safety requirement is falsified (comparable to that a FTA top-level event occur), a MCS of order 1 is found, consisting of the executed fault combination. This fault combination is then removed from the set, thus prohibiting it from being used in future experiments. The evaluation then continues until all combinations have been tested. Remaining combinations in the set are then combined into fault combination pairs that are injected into the model in order to explore its robustness against double faults. This then continues until the combination set is empty (i.e. all MCS have been found). In practice however, it might be desirable to end the execution at \( n \) simultaneous faults due to plausibility concerns. Algorithm 1 shows a pseudo-code implementation of the fault injection based MCS generation algorithm for Simulink models. FMFs implement the effects of faults that are inserted into the model. These can e.g. be bit flip faults or signal amplification (gain) faults. In order to use these FMFs, additional parameters need to be set, e.g. which bit to flip or the value of the gain fault (e.g. 20x or 2000x). This is not visible in the algorithm, but is assumed to be part of the FMF/Location pair input. It is often desirable to perform experiments with several different values, as shown in Figure 4-14.
Function \texttt{GenerateMinimalCutSets()}

\textbf{Inputs:}
- \texttt{MCSSizeMax}: Maximum MCS Size
- \texttt{Model}: System behavior model
- \texttt{Reqs}: System requirements
- \texttt{FMFLocList}: List of FMF/Location pairs
- \texttt{Stimuli}: Behavior model input port stimuli
- \texttt{Timebase}: Simulation time information

\textbf{Outputs:}
- \texttt{MCSList}: List of Minimal Cut Sets

\textbf{Sub Functions:}
- \texttt{FindCombinations(M, T, MCSList)}: Find all combinations of \textit{M} FMF/Location/Time tuples from \texttt{FMFLocList} and \textit{T}, excluding those which include combinations that are stored in the \texttt{MCSList}.
- \texttt{TransformModel(Model, C)}: Insert FMF to locations defined in \textit{C} into \texttt{Model}.
- \texttt{Run(Model, Stimuli, Time, Reqs)}: Perform simulation or calculate model outputs with faults enabled during time steps defined in \texttt{Time}. Provide input ports with data from \texttt{Stimuli}. If any requirement in \texttt{Reqs} is falsified, return FAIL otherwise return PASS

\textbf{Algorithm:}

\begin{verbatim}
Clear MCSList
FOR M = 1 to MCSSizeMax
    FOR ALL Time Combinations T from Timebase
        myCList = FindCombinations(M, T, MCSList)
        FOR ALL items C in myCList
            myModel = TransformModel(Model, C)
            Verdict = Run(myModel, Stimuli, T, Reqs)
            IF (Verdict == FAIL)
                Add (C,T) to MCSList
            END IF
        END FOR
    END FOR
END FOR
Return MCSList
\end{verbatim}

Algorithm 1: Fault Injection Based Minimal Cut Sets Generation

Consider the system depicted in Figure 4-6 which is a Simulink behavior model of a Triple Modular Redundancy (TMR) pedal sensor voter model. It takes as input three sensor values (0 – 65535) and outputs the sensor value after voting.
By an analysis of the TMR sensor voter it can be argued that the small system allows one pedal sensor to deviate from the others without affecting the functionality and violate the safety of the system, i.e. the system is single fault tolerant against single faults on input signals. Further analysis of the model reveals that the system is tolerant against single faults except when they occur in $\text{Switch 1}$. We will however walk through Algorithm 1 for this small example. The algorithm requires four input parameters which we provide as:

$\text{MCSSizeMax}: \ 2$, i.e. we investigate tolerance of single and double faults

$\text{Model}: \ \text{The model in Figure 4-6}$

$\text{Reqs}: \ \text{Requirement “Output pedal value must not differ from the output pedal value obtained during the fault free run”}$

$\text{FMFLocList}: \ \text{List of all FMF/Location combinations to be investigated, see Table 4-1.}$

$\text{Stimuli}: \ \text{See below}$

$\text{Timebase}: \ \text{Simulation time steps: \{0,1,2,3\}}$

In this example, only a subset of all FMF/Location combinations is shown in order to simplify the description of the algorithm.

<table>
<thead>
<tr>
<th></th>
<th>Stuck-At-Zero (F1)</th>
<th>Gain (2x) (F2)</th>
<th>Bit flip (Bit 5) (F3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pedal Sensor 1 (L1)</td>
<td>o</td>
<td>o</td>
<td>N/A</td>
</tr>
<tr>
<td>Pedal Sensor 2 (L2)</td>
<td>o</td>
<td>o</td>
<td>N/A</td>
</tr>
<tr>
<td>Pedal Sensor 3 (L3)</td>
<td>o</td>
<td>o</td>
<td>N/A</td>
</tr>
<tr>
<td>Switch 1 (L4)</td>
<td>N/A</td>
<td>N/A</td>
<td>o</td>
</tr>
<tr>
<td>Switch 2 (L5)</td>
<td>N/A</td>
<td>N/A</td>
<td>o</td>
</tr>
</tbody>
</table>

Table 4-1: Applicable FMFs for different Locations

Table 4-1 shows a matrix where the applicability of FMFs is shown for some locations in the example system. FMFs (F1) and (F2) are applicable to real values (floating-point numbers) and integers and are therefore suitable for external values outside of the modelled system (e.g. sensor values) while FMF (F3) is a bit-level FMF, thus applicable mainly to system internal variables (e.g. Switch 1 and Switch 2 in the example). In this example, the Switch 1 and Switch 2 output values as well as the pedal sensor values are represented by 16-bit unsigned integers.

The FMFLocList variable consists of the (FMF/Location) pairs: {F1,L1}, {F2,L1}, {F1,L2}, {F2,L2}, {F1,L3}, {F2,L3}, {F3,L4}, {F3,L5} which is derived from the table.

In order to simulate the model behavior it is necessary to provide input port stimuli (i.e. sensor values for pedal sensors in this particular example). For this example it is:

Pedal sensor 1 = Pedal sensor 2 = Pedal sensor 3 = \{0, 600, 32768, 65535\} for the four time steps \{T1,T2,T3,T4\}.

Before we execute the algorithm, we need to create Golden Run data, i.e. output port values (Pedal Value) during the fault free run. In this example, the output value will be \{0, 600, 32768, 65535\} for the four time steps. These output values are then used as oracle for the calculation of experiment verdicts.

After the first round (i.e. M = 1), the MCSList will contain the following combinations:

MCSList = \{F3,L4,T1\}, \{F3,L4,T2\}, \{F3,L4,T3\}, \{F3,L4,T4\} (i.e. bit-flip fault on Switch 1 falsifies the requirement for all 4 time steps). This implies that the combination \{F3, L4\} will be excluded during round 2 (M = 2). For the second round however, the list of combinations returned by FindCombinations should be:

\{(F1,L1), (F1,L2), (F1,L3), (F1,L4), (F1,L5)\}. Notice the vast amount of combinations occurring even in this small example. The combinations above are only a small subset from the
total of 18 combinations, assuming that only one fault can be injected on each location simultaneously. This means that a significant amount of experiments are necessary to perform in order to cover all combinations. Taking into account the time domain with all its combinations, the number of combinations grows rapidly. Upon completion of the algorithm, we will have a list (MCSList) of all combinations that falsifies the requirement.

4.5 Test Cases Developed From Minimal Cut Sets

Generated MCS are useful in different phases of development. During early development phases, the experiment results can be used to find fault sensitive designs which require redesign and/or addition of error detection mechanisms (EDMs) and error recovery mechanisms (ERMs). In the previous TMR sensor example, it might mean a redesign of the model to provide dual outputs that are fed to another system instead of the single one leading to a single point of failure, thus making the model single fault tolerant against the investigated fault models. To find fault sensitive parts of the system and add EDMs and/or ERMs into the model instead of adding them later generally saves time and effort both during development as well as during testing. The EDMs and ERMs can also be automatically generated as source code from the Simulink behavior model and can be verified through fault injection experiments both in Simulink as well as with physical fault injection on the target system.

In later phases when new versions of a system are created, the generated test cases can be used for regression testing to ensure that old problems do not re-appear and that new bugs are not introduced. Figure 4-7 shows an example of a fault injection test case which could serve as input for physical fault injection. It contains information about the faults that are injected as well as the requirements that are falsified. In this example a double fault is simulated by injecting failure mode function 15 (i.e. value stuck at zero) as defined in MOGENTES D3.1b [D3.1b, 2009] into an input port block and failure mode function 30 (i.e. bit flip fault) into Switch 2. The faults are then enabled during a single time step (t = 0.1), thus simulating a transient fault. In addition, it also contains the nominal output signal values as well as input port stimuli.

```xml
<MODEL-STIMULI>
  <TIME-STEP>0</TIME-STEP>
  <TIME-STEP>0.1</TIME-STEP>
  <TIME-STEP>0.2</TIME-STEP>
  <TIME-STEP>0.3</TIME-STEP>
</TIME-STEP>

<STIMULI-DATA/>

<PORT-STIMULI>
  <PORT-REF>ThrottleModel/In1</PORT-REF>
  <PORT-DATATYPE>uint16</PORT-DATATYPE>
  <VALUE>0</VALUE>
  <VALUE>660</VALUE>
  <VALUE>32768</VALUE>
  <VALUE>65535</VALUE>
</PORT-STIMULI>

<PORT-STIMULI>
  <PORT-REF>ThrottleModel/In2</PORT-REF>
  <PORT-DATATYPE>uint16</PORT-DATATYPE>
  <VALUE>0</VALUE>
  <VALUE>660</VALUE>
  <VALUE>32768</VALUE>
  <VALUE>65535</VALUE>
</PORT-STIMULI>

<PORT-STIMULI>
  <PORT-REF>ThrottleModel/In3</PORT-REF>
  <PORT-DATATYPE>uint16</PORT-DATATYPE>
  <VALUE>0</VALUE>
  <VALUE>660</VALUE>
  <VALUE>32768</VALUE>
  <VALUE>65535</VALUE>
</PORT-STIMULI>

</MODEL-STIMULI>

<MODEL-STIMULI>

</MODEL-STIMULI>

<OUTPUT-PORT-DATA>
  <PORT-REF>ThrottleModel/Pedal Value</PORT-REF>
  <VALUE>0</VALUE>
  <VALUE>660</VALUE>
  <VALUE>32768</VALUE>
  <VALUE>65535</VALUE>
</OUTPUT-PORT-DATA>
```
<GOLDEN-RUN-DATA>
<EXPERIMENT-RESULTS>
<EXPERIMENT-RESULT>
<EXPERIMENT-SUCCESSFUL>true</EXPERIMENT-SUCCESSFUL>
<EXPERIMENT-FAILURECOMMENT>
<VIOLATED-ASSERTION>
<VIOLATED-MATH-REQUIREMENT>
((y(1)-yG(1))==0)
</VIOLATED-MATH-REQUIREMENT>
</VIOLATED-MATH-REQUIREMENTS>
</EXPERIMENT-FAILURECOMMENT>
</EXPERIMENT-SUCCESSFUL>
</EXPERIMENT-RESULT>
</INJECTED-FAULTS>
</INJECTED-FAULT>
</INJECTED-FAULTS>
</OUTPUT-PORT-DATA>
</OUTPUT-PORT-DATA>
</EXPERIMENT-RESULTS>
</GOLDEN-RUN-DATA>

Figure 4-7: XML example of a fault injection test case

### 4.6 Test Case Generation with SP MODIFI

SP MODIFI is a fault injection tool that is able to inject faults into MATLAB/Simulink models. The fault injection tool can perform fault injection on any Simulink block output signal, i.e. even on complex blocks such as state machines or blocks where the logic is implemented in code (mex-functions). SP MODIFI uses a model implemented fault injection approach. This means that fault manifestations are modelled into standard Simulink blocks that are actually inserted into the behaviour model under analysis and activated during simulation as depicted in Figure 4-1. This fault injection tool supports injection of permanent faults as well as transient faults, i.e. faults that occur during one time instant.

![Figure 4-8: Transient fault](image-url)
Transient faults do not cause permanent hardware damage, but may lead to bit-flips in microcontrollers or other logic components. The effects of transient faults can be hard to predict and is often not considered in traditional analysis methods such as FTA or FMEA. SP MODIFI supports the injection of permanent faults to perform an automated FMEA (Failure Modes and Effects Analysis). This is however not a unique feature, but can also be done with other tools [AUTOFMEA, 2009]. SP MODIFI implements Algorithm 1 (which is described in section 4.4) and is therefore able to create MCS based fault injection test cases that can be used for physical fault injection as well as for regression testing and model improvements.

SP MODIFI is controlled using a Java-based Graphical User Interface (GUI) which is started from within MATLAB, while the fault injection engine is implemented in MATLAB code. The data information exchange format that is used in the communication between the GUI and the fault injection engine, as well as for the experiment logging, is XML. This makes it suitable for information exchange with external tools (e.g. the SP MODIFI result analysis tool which is a C# based stand-alone tool for offline analysis of the fault injection experiments).

![Figure 4-9: SP MODIFI Graphical User Interface; Step 1](image)

The fault injection experiment setup in the GUI (Figure 4-9) requires the user to load a Simulink model file together with a stimuli file. The stimuli file is used for two purposes: to define the simulation time and to supply signal values for all input ports in the entire system for the duration of the simulation. The user is also able to supply a path to a MATLAB .m-file that shall be executed before a simulation is performed. This is useful e.g. when certain global parameters must be set before the simulation may be performed. Finally, the user pushes the button “Verify Settings”, which starts the simulation of the system without faults injected to verify that the stimuli file is valid for the given system and that sufficient information was provided in the Pre-injection execution MATLAB-file. When the verification is complete, the user is able to continue the configuration.
Figure 4-10: SP MODIFI Graphical User Interface; Step 2

The next step in the configuration is to perform part of the pre-injection analysis, i.e. a reduction of the virtually infinite experiment space into a manageable set of fault injection experiments. This is done by defining applicable fault models for each of the signals in the behaviour model that shall be subject to fault injection. Figure 4-10 shows how the behaviour model is represented in a tree view where the user navigates to perform the pre-injection analysis. The user can also define whether signals will be internal (i.e. internal to a microcontroller by using memory or registers) or external (e.g. sensor signals or actuator control signals). This has an impact upon the fault models applicable for a given signal. For internal signals, only bit-level fault models (i.e. bit flip or bit stuck at) might make sense. For external signals, other fault models are applicable, depending on the signal representation (e.g. Boolean or Real). An attempt to automatically determine whether the signal will be internal or external in the final representation is done by the tool. This is based upon the hierarchical structure (i.e. root level ports are assumed to be interfaces to surrounding environment) but can easily be overridden by the user to represent the true system implementation.
The subsequent step in the configuration is to set up requirements (Figure 4-11). After an experiment has been performed, it is evaluated to ensure that all requirements are met. If not, it will be indicated since it is a proof that the requirement can be falsified by using fault injection. It is not mandatory to use requirements for the fault injection experiments. In some cases it might be useful to perform fault injection experiments without evaluating the outcome, e.g. when analysis is to be performed in an external tool. SP MODIFI can currently handle two ways of describing requirements. Requirements can be modelled into the behaviour model by using the standard Simulink block “Assertion”. This is useful when requirements are depending on e.g. internal variables, states or when complex requirements are described, e.g. requirements that are dependant on more than the current value of a behaviour model system output variable. Figure 4-11 shows a screenshot of a system that have three requirements described in the behaviour model. In the GUI the user can select which of them to activate during the experiments. The other way to describe requirements in SP MODIFI is to use the requirement editor. This is useful for requirements that involve current values of behaviour model output values, e.g. a value must not deviate from its nominal behaviour or must not exceed a certain threshold. The nominal behaviour is gathered during a fault free run, a golden run, which is used to compare the fault injection experiment results against. In Figure 4-11 one such requirement is added: “Output signal value x must not differ from its nominal behaviour”.

Figure 4-11: SP MODIFI Graphical User Interface; Step 3
The next step in the fault injection experiment setup is to set up a few parameters in the “Experiment Settings” tab. These are divided into Window Settings and Simulation Settings. Under the Window Settings the user can define when to observe the system behaviour and compare to the nominal run (Observation Window) and when to inject faults (Fault Injection Window). Since the systems analyzed in this tool are causal, it makes no sense to observe the system behaviour before any faults are injected, which saves execution time. It also makes no sense to inject faults after the observation window has elapsed. The window settings can be useful to prohibit the tool from injecting faults e.g. before the system has been simulated into a certain state.

Under Simulation Settings, the user provides the final settings of the fault injection experiments. Experiment Timeout is used to force an abortion of the simulation if it exceeds the defined timeout time. During certain fault injection experiments the simulation time converges towards infinity due to e.g. singularities. The Campaign Timeout setting defines the overall timeout for all experiments. Due to the virtually infinite search space it is in most cases impossible to do an exhaustive analysis. The Max Minimal Cut Set Size parameter defines the maximum number of simultaneous faults that shall be injected during one fault injection experiment. The fault injection engine starts by injection single faults only, and then continues to inject double faults for all faults that did not falsify any requirement in the single fault case. The Time-steps per FMF/Location Pair parameter defines how many transient fault injection experiments that shall be performed for every combination of fault model and location (signal). This is in addition to the permanent fault injection experiment that is performed for every fault model/location combination. The faults are then injected during a single simulation step starting at time \( n \) which is a time between the minimum fault injection window and the maximum fault injection window. The final setting is the Collapse Time Domain flag, which tells the fault injection engine to cancel any further experiments for the same combination of fault model and location, i.e. ignoring that the same combination of fault model and location can lead to different system behaviour depending on when they are injected. This is often useful in systems without state transitions.
Figure 4-13 shows the final tab of the fault injection experiment setup. In this tab, the user only need to select where the output log XML-file shall be stored and then either export the settings for future/other use or command the fault injection engine to perform the experiments. The user can then visually follow the fault injection experiment progress in a progress window (Figure 4-14). The progress window shows one cube for each fault model that is investigated. The three axes on a cube represent the locations (signals), value (e.g. bit number for a bit level error or the gain value for an amplification error) as well as the positions in time when faults are injected into the behaviour model. Each position in the cube represents an experiment, and the user can visually distinguish between experiments to be performed, experiments that were unsuccessful (e.g. due to timeout), experiments that were successful but falsified requirement(s) and experiments that were successful and did not falsify any requirement.
When the fault injection experiments have been performed, the log file is a refinement of the configuration XML-file i.e. the experiment setup is stored together with verdicts. This makes it possible to re-use the log file for regression testing. The resulting log-file can now be used in various ways, e.g. for refinement into test cases for physical fault injection or for analysis of the current model to make it more robust.

Figure 4-15 shows a screenshot of the SP MODIFI analysis tool. This tool can be used for analysis of the generated log files.

4.7 Failure Mode Function Models

XML is used as a common description language to describe the FMFs proposed in the MOGENTES deliverable D3.1b. Thus, different tools can use the same general description of hardware fault models (failure modes). A representative set of the FMFs were selected, their pseudo code implementations studied,
and based on those, draft XML descriptions were handcrafted. The set of XML descriptions where investigated for commonalities and a supporting XML schema was established, see Figure 4-16. (The change operator FMF and communication FMFs have not been considered (most likely the communication FMFs can at least partly be implemented using the basic FMFs))

Figure 4-16 Graphical representation of the XML schema for the description of FMFs

Each FMF is recognized by its name and a unique id. All FMFs have an input trigger port and at least one input and one output port. Many FMFs also use constants and parameters for their operation. Ports, constants, and parameters have names and associated datatypes. The ports also have a direction. Every FMF has at least one trigger condition, i.e. a condition related to the value and timing of the input trigger port and possibly to the input and output ports as well. Associated to each trigger condition is an operation consisting of one or several commands. The main task of the commands is to transform the nominal value into a faulty value, e.g. by adding constant value, and presenting it at the output of the FMF.

An example FMF, one of the simplest, is shown in Table 4-2 and its XML description is shown in Figure 4-17. It is an FMF operating on Boolean values and when it is active, i.e. the input trigger is TRUE, the output is forced to the FALSE value.

Table 4-2 Example FMF

<table>
<thead>
<tr>
<th>Group</th>
<th>FMF</th>
<th>Pseudo code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Off (ID=2)</td>
<td>FMF_Bo_Off()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if fi_trigger[n]==TRUE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>output[n] = FALSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td></td>
<td>output[n] = input[n]</td>
</tr>
</tbody>
</table>
The example FMF has one input, one trigger, and one output port. It has two Boolean constants and no parameters. It has a single trigger condition which checks if the trigger signal is \textit{TRUE} at the specific time instant (no delay). In the associated operation the output port is assigned a constant Boolean \textit{FALSE} value.
5 Prover iLock Based Test Case Generation

This section describes the Prover iLock-based track in MOGENTES related to generation of minimal cut sets based on counter examples to formal verification of safety requirements, and generation of test cases based on those counter examples and minimal cut sets. This method for MCS generation differs from the method described in section 4.4 in several ways: the application domain area is different (railway interlocking systems versus more general-purpose systems), the languages used for representing the system model, and in particular the way in which the minimal cut sets are identified. In this track, the minimal cut sets are based on top events being failing safety requirements.

This section also provides a brief description of the overall Prover iLock process, related to the MOGENTES prototype for fault injection and minimal cut set generation, with experimental results achieved.

5.1 Introduction

Formal verification is a technique that can be used for demonstrating, using formal proof, that a system satisfies a set of properties. Typical properties that are verified using formal verification are safety properties, stating some desired property that the system should always satisfy (alternatively, a safety property expresses unwanted behaviour that the system should never allow).

The result of the formal verification of a requirement is one of the following:

- If a safety requirement is proved, it is called *Valid*.
- If a safety requirement fails, it is called *Falsifiable*, and a *counter-example* that demonstrates an execution trace of the system leading to a state that violates the requirement is returned.
- In case a safety requirement cannot be proved Valid nor shown to be Falsifiable (because it is too computationally challenging), it is called *Indeterminate*.

Given a system model with a set of properties that can be proved Valid (this is called the *Nominal Analysis*), we can extend the system to allow faults (one fault, two faults, and so on) to take place, and apply formal verification of the properties (again). If we can prove a property Valid with a set of faults allowed, the faults that were allowed could not be observed with respect to the validity of the property in question. If a property becomes Falsifiable with faults active, it is known that the faults caused the property to fail. The faults that are active in the counter example to the failing property is a cut set.

Allowing increasingly many faults to be active in the formal verification process, we get a method (an algorithm) to compute Minimal Cut Sets (MCS). A minimal cut set is a minimal set of faults that cause a top event; in this case the top event is a Falsifiable property.

In MOGENTES, generation of minimal cut sets using formal verification is applied in the Prover iLock track, in which it is applied on a railway demonstrator system (an interlocking system) created in Prover iLock. The following sections provide more detail on the minimal cut set generation and its application.

5.1.1 Process

The process for computing minimal cut sets consists of the following three main steps:

- Nominal analysis – formally verify that a system model satisfies a set of properties
- Inject faults into the system model
- Apply the minimal cut set generation algorithm (of section 5.1.7) to the fault injected system model
- Convert the minimal cut sets into test cases

In the Prover iLock-based track in MOGENTES, the system model, the safety properties, the fault injection and the minimal cut set generation are applied from the Prover iLock tool suite. The overall usage process is illustrated by Figure 5-1.
A layout configuration provides the configuration of the Specific Application (a computerized interlocking system).

Based on the Layout configuration and the Generic Application (generic specifications of a family of interlocking systems in the language of PiSPEC) the design and safety requirements for the Specific Application are generated.

Nominal analysis is performed to prove that the design satisfies its safety requirements.

Faults are injected into the design, effectively replacing a subset of the variables in the design with new definitions of variables (including faults).

MCS generation is performed, creating a report of the minimal cut sets identified, as well as creating a set of test cases based on the minimal cut sets identified.

The design with faults is made subject to code generation, exporting it as C code. The generated C code includes the safety requirements as a "safety bag" (for the validation step below).

The test cases exported are validated by checking that the test cases do lead to failing safety properties in execution of the C code; the C code includes a safety bag to enable verification of that the MCS-based test cases do yield the violation of safety properties.

The above steps have been implemented in a prototype around the Prover iLock tool suite, as described in sections 5.1.2 through section 5.1.7. Section 5.1.8 describes experimental results that have been achieved using a Prover iLock demonstrator of a generic interlocking system design.

5.1.2 Design Creation

This step creates a design of a computerized interlocking system (CBI) based on instantiation of a Generic Design Specification (GDS) and the station-specific application configuration:
• The GDS defines generic signaling principles common to a family interlocking systems. The GDS is define in the PiSPEC language, which is a relatively simplistic object-oriented language for boolean equations including predicate logic (to enable expressing generic equations).

• The station-specific configuration consists of the station’s track layout, defining the geographical arrangement of signals, switches and other objects, as illustrated in the screenshot in Figure 5-2. The station-specific configuration can also define virtual objects, such as routes and overlaps.

The design is created by partial evaluation of predicate logic expressions in the GDS, which instantiate the GDS based on the station-specific configuration.

The design is a set of ordered equations, where the left-hand-side is a Boolean variable and the right-hand-side is a Boolean expression. Expressions combine constants, variables and inputs using Boolean connectives (AND, OR, NOT...). Equations can have associated delays, delaying an assignment to value true or false.

Figure 5-3 is a screenshot from Prover iLock, showing an assignment to variable TC21.unblock. The expression is displayed graphically in the upper right, and textually in the lower right. The left part shows a list of variables in the design.

Figure 5-2: Prover iLock track overview
5.1.3 Requirements Creation

This step creates the safety requirements for a computerized interlocking system (CBI) based on instantiation of a Generic Safety Specification (GSS) and a set of rules that based on the station-specific application configuration and the design determine the requirements to create.

The GSS defines generic safety requirements common to a family interlocking systems. The GSS is defined in the PiSPEC language, supporting a relatively simplistic object-oriented language of boolean properties including predicate logic (to enable expressing generic properties).

Programmable rules instantiate classes in the GSS based on the station-specific configuration, producing specific requirements (instances of classes containing requirements) for the specific system.

A requirements is a named safety requirement, defined as a Boolean expression. Expressions combine constants, variables and inputs using Boolean connectors (AND, OR, NOT...) and can use the temporal operator PRE (referring to the previous value of an expression).
As shown in Figure 5-4, it is also possible to visualize each step in the sequence using the graphical representation of the model.

5.1.4 Formal Verification (Nominal analysis)

In nominal analysis, the system is formally verified to satisfy its safety requirements using Prover iLock Verifier. The formal verification is carried out by the built-in model-checker.

- If a safety requirement is proved, it is Valid.
- If a safety requirement fails, it is Falsifiable, and a counter-example that demonstrates an execution of the system leading to a state that violates the requirement is returned.
- In case a safety requirement cannot be proved Valid nor shown to be Falsifiable (because it is too computationally challenging), it is Indeterminate.

Figure 5-5: Prover iLock verification result
Figure 5-5 shows the result of a formal verification session in Prover iLock Verifier. All safety properties are Valid, except for requirement K.H which is Falsifiable. The user can inspect the counter example (values on inputs and other variables) to debug the cause of the failure (the right panel in Figure 5-5). Figure 5-6 shows an alternative view for debugging, in which part of a counter example is inspected using a graphical ladder logic diagram.

**Figure 5-6: Graphical step visualization**

### 5.1.5 Fault injection

Fault injection applies a generic rule for inserting faults into the system model. In order to compute relevant minimal cut sets, the model must faithfully reflect the behavior of the modeled system under the influence of failures. This can be achieved in two different ways:

1. Faulty behavior is explicitly modelled in the basic blocks which are used to create the system, or
2. An existing model is modified to reflect the effect of faults.

Prover iLock supports both these approaches. Using the second approach, the original model can be modified in an automated manner. This modification can be seen as a mutation of the original design model. This design mutation is created by specifying the signals (variables) in the model that correspond to components that can fail, and then providing the failure mode to apply for each signal ("mutating" the signal). This can be done for each component individually, or for ranges of components, depending on component types.

In Prover iLock, the fault injection step can be done using a programmable rule, implemented as a Python script that invoked by the user.

For instance, one may specify that a variable TC21.M_unblock corresponds to a sensor that is part of the model and is subject to an "invert" failure mode, where the value which it reports is the opposite of the value it should report. The modified version of the model introduces a new Boolean input variable, TC21.M_unblock_FAIL, indicating if the sensor is failing (active). Another Boolean variable is introduced, named "TC21.M_unblock_COPY", which is assigned "TC21.M_unblock XOR TC21.M_unblock_FAIL", thus modeling the "invert" failure mode. All occurrences of TC21.M_unblock in the original model are replaced by TC21.M_unblock_COPY in the new model. Naturally, more complex failure modes than this example can be used.
5.5.1.1 Faults

Each fault is assumed to introduce a fresh boolean input. If this input is true, it activates the fault. We call this input the **fault activation condition**.

The MCS computation (see section 5.1.7) does not care about the type of fault, but rather only cares about the fault activation condition. The MCS computation algorithm considers a fault to be active if its fault activation condition is true at least once during an execution trace.

Currently, there is no file format used for faults. Instead, the fault function is part of the programmable rule that performs the fault injection. It would be straightforward to add support for a fault file format, should one want to.

5.1.6 Create verification model

In this step, the verification model for the MCS computation is created. The verification model is created based on the system model containing faults. The verification model is on a file format that can be read by the proof engine of Prover iLock Verifier.

5.1.7 MCS computation Algorithm

The MCS computation program is a prototype that takes a verification model and an MCS level parameter as input, and returns the set of MCSs that were computed upto that level. The prototype is implemented as a Python program that currently is invoked outside of Prover iLock. The MCS computation algorithm uses an Application Programming Interface (API) to the proof engine underlying Prover iLock Verifier.

Each minimal cut set has an associated counter example for the failed safety requirement(-s). The counter example is a trace of input variable values, including the fault activation conditions for all faults, leading to a state that violates the safety requirement.

The following outlines the MCS computation algorithm.

**Formal verification-based MCS generation algorithm**

```python
def add_mcs_constraint(req, l):
    """Exclude the accumulated set of MCSs up to MCS level l for requirement req"""

def constrain_cardinality(n, variables):
    """Enforces n number of variables to be true in execution traces of the system"""

def prove(reqs):
    """Perform model checking and return the set of proved/falsified requirements"""

def active_faults(p):
    """Return true fault activation conditions in counter example to requirement p"""

def r.add_mcs(l, faults):
    """Add an MCS of level l to requirement r"""

def mcs(level, faults, REQS):
    for mcs_level in 1..level:
        falsified = 0
        iterations = 0
        active_REQS = REQS
        while (iterations == 0 or falsified > 0):
            iterations = iterations + 1
            for po in active_REQS:
                add_mcs_constraint(po, mcs_level)
                constrain_cardinality(mcs_level, faults)
            (proved_REQS, falsified_REQS) = prove(active_REQS)
            for r in proved_REQS:
                active_REQS := active_REQS \ r
            falsified = 0
            for r in falsified_REQS:
```
falsified = falsified + 1
r.add_mcs ( mcs_level, active_faults(r))

Note that the MCS computation algorithm considers a fault to be active iff the fault occurs at least once during an execution trace.

The implementation is somewhat naive in its current form – for instance, it may be worthwhile to tune the proof strategies used (in the proof engine) as well as trade exactness for speed in an overall heuristics.

5.1.8 Example experiments

The MCS computation prototype has been experimented with using the Prover iLock demonstration platform for MOGENTES. The demonstration platform contains a generic design and a set of associated generic safety requirements developed for a fictitious family of interlocking systems. The signaling principles and the safety requirements are inspired by Swedish signaling and regulation, albeit reduced to a simpler form than one would find in a real industrial application. A simple meeting station was used as example. In fact, the station is visualized in the previous screenshot of Figure 5-2.

Experiment 1

The following table provides statistics for MCS computation of levels 1 through 4 for the MOGENTES example system. In this experiment, the same type of fault had been injected on every system input (in total 174 input variables). The injected fault causes the opposite value to be read for each input.

The first row describes the nominal analysis (with no active faults), followed by a row for each level of the MCS computation.

- One of the requirements was falsified in the nominal analysis, and hence not included in the subsequent MCS computation
- Only two requirements were sensitive to the faults that had been injected
- Only MCS of size 1 were generated; there are no MCS of size 2, 3 or 4
- While the system is quite small compared to a realistic interlocking system implementation, the overall MCS computation of levels 1 to 4 took about 1900s in total (using a regular laptop PC).

<table>
<thead>
<tr>
<th>MCS</th>
<th>No. of falsified requirements</th>
<th>No. of MCS</th>
<th>No. of iterations</th>
<th>Computation time</th>
<th>No. of counter examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal analysis</td>
<td>1/10</td>
<td>N/A</td>
<td>1</td>
<td>85s</td>
<td>1</td>
</tr>
<tr>
<td>MCS size 1</td>
<td>2/9</td>
<td>24</td>
<td>17</td>
<td>1500s</td>
<td>24</td>
</tr>
<tr>
<td>MCS size 2</td>
<td>0/9</td>
<td>0</td>
<td>1</td>
<td>100s</td>
<td>0</td>
</tr>
<tr>
<td>MCS size 3</td>
<td>0/9</td>
<td>0</td>
<td>1</td>
<td>112s</td>
<td>0</td>
</tr>
<tr>
<td>MCS size 4</td>
<td>0/9</td>
<td>0</td>
<td>1</td>
<td>70s</td>
<td>0</td>
</tr>
</tbody>
</table>

Experiment 2

The following table provides the same type of statistics and performance data for MCS computation for the same system with the same MCS levels, but in this experiment the fault function (applied on each input signal) was changed to value false if the fault activation condition is true.

- The same two requirements were sensitive to the faults that had been injected
- Only MCS of size 1 were generated (in total 16 MCSs); there are no MCS of size 2, 3 or 4
• The overall MCS computation of levels 1 to 4 took about 1200s in total (using a regular laptop PC).

<table>
<thead>
<tr>
<th>MCS</th>
<th>No. of falsified requirements</th>
<th>No. of MCS iterations</th>
<th>Computation time</th>
<th>No. of counter examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal analysis</td>
<td>1/10</td>
<td>N/A</td>
<td>85s</td>
<td>1</td>
</tr>
<tr>
<td>MCS size 1</td>
<td>2/9</td>
<td>16</td>
<td>800s</td>
<td>16</td>
</tr>
<tr>
<td>MCS size 2</td>
<td>0/9</td>
<td>0</td>
<td>100s</td>
<td>0</td>
</tr>
<tr>
<td>MCS size 3</td>
<td>0/9</td>
<td>0</td>
<td>100s</td>
<td>0</td>
</tr>
<tr>
<td>MCS size 4</td>
<td>0/9</td>
<td>0</td>
<td>70s</td>
<td>0</td>
</tr>
</tbody>
</table>

5.2 Summary

Formal verification-based minimal cut-set generation considers all possible execution paths of the system (by the use of a complete formal verification method). If a fault in the system model can cause a system property to fail, a counter example trace that provides values to the system’s variables over time is generated. This makes it possible, in theory, to apply exhaustive analysis of the safety impact of faults using minimal cut sets. Generation of minimal cut sets is an important tool for assessing the fault-tolerance of a system, and therefore of high relevance to safety-critical systems whose safety depend on components (sensors, inputs, internal components and so on) that can fail.

In our experiments performed, it has been rare to discover MCS of degree higher than one. While such examples exist (and have been manually created as part of this work), we speculate that these are rare and of less relevance for analysis of computerized interlocking systems. Our experiments have been limited to apply faults on system inputs, since this is the most relevant fault to consider for a computerized interlocking system. We plan to extend our experiments by applying MCS generation also for relay-based interlocking systems, in which it can be more relevant to consider a larger set of possible faults; physical components such as relays, diodes etc can fail, and it is anticipated that interesting analyses will result for typical relay-based interlocking systems.
6 Abbreviations and Definitions

EDM  Error Detection Mechanism
EMI  Electro Magnetic Interference
ERM  Error Recovery Mechanism
ESM  Extended System Model
FI   Fault Injection
FMF  Failure Mode Function
FTA  Fault Tree Analysis
HIFI Hardware-Implemented Fault Injection
IEC  International Electrotechnical Commission
IOLTS Input Output Labeled Transition System
ISO  International Organization for Standardization
MCS  Minimal Cut Set
MIFI Model-Implemented Fault Injection
QAS  Qualitative Action System
TMR  Triple Modular Redundancy
SWIFI Software-Implemented Fault Injection
VHDL Very High Speed Integrated Circuit Hardware Description Language
7 References


[D3.1b, 2009] MOGENTES deliverable D3.1 *Fault Models (first version)*


