xSAP
The eXtended Safety Assessment Platform
Version 1.4

xSAP User Manual

by
Fondazione Bruno Kessler
Embedded Systems Unit

Contents

Chapter 1

Introduction

The design of complex systems requires the ability to analyze, in additional to functional correctness, also the way faults are dealt with. This approach, known as safety assessment, relies on the definition of possible faults of a system, and results in the construction of important artifacts such as Fault Trees and FMEA tables.

Purpose of this document is to illustrate the usage and underlying principles of xSAP (eXtended Safety Assessment Platform). xSAP is a tool for the formal analysis and safety assessment of (complex) systems. xSAP supports a formal approach, based on principles of symbolic model checking. With respect to a model checker, xSAP provides the following functionality:

Fault Extension: starting from a *nominal* model, describing the behaviour of a system without faults, an *extended* model is produced, that also contains behaviours in the presence of faults. Such extended model can be modeled manually (*manual fault extension* – compare Section 3.1) or automatically, starting from a library of faults and a definition of the faults to be injected into the nominal model (*automated fault extension* – compare Section 3.2).

Fault Tree Analysis: given an extended model, and a specific property, xSAP produces the fault trees collecting all the minimal cut sets that can result in a property violation. This technique is classified as a *top-down* analysis – compare Section 4.2.

FMEA tables: given an extended model, xSAP produces FMEA tables, describing which properties are violated as a consequence of the occurrence of which faults. This technique is often referred to as a *bottom-up* technique, proceeding from the faults to the top properties – compare Section 4.3.

At the core, xSAP relies on the nuXmv model checker [15], from which it inherits the modeling language. Within xSAP, the functional verification capabilities of nuXmv are extended to deal with model extension and safety assessment.

xSAP is designed to provide the following advantages. First, xSAP supports a methodology that allows for a tight integration between the design and the safety teams. Second, it automates (some of) the activities related both to the verification and to the safety analysis of systems in a uniform environment. Third, the use of the platform is compatible with an incremental development approach, based on iterative releases of the system model at different levels of detail.

History of xSAP xSAP is developed and maintained by the Embedded Systems Unit of FBK. xSAP is a reimplementation of the FSAP/NuSMV-SA platform. FSAP/NuSMV-SA (Formal Safety Analysis Platform) was developed since 2001, with the support of the European Union, within several projects in the areas of formal verification and safety analysis: the ESACS project, the ISAAC Project, and the MISSA Project. The techniques underlying FSAP/NuSMV-SA have been subsequently applied in several projects funded by the European Space Agency: COMPASS[1], AUTOGEF[2], FAME[3], and HASDEL[4].

Structure of the document This manual is organized as follows:

- Chapter 2 describes the overall process supported by xSAP.
- Chapter 3 explains the Fault Extension methodology.
- Chapter 4 presents the available forms of safety analysis.
- Chapter 6 gives and overview of the Fault Detection and Isolation analysis.
- Chapter 7 shows a complete example of use.
- Chapter 8 draws some conclusions and outlines the directions of future development.

The appendix contains the following information:

- Chapter A specifies the necessary hardware/ software configuration needed to run the xSAP toolset and the required installation steps.
- Chapter B describes the Syntax Directed Editor for the fault extension instruction language.
- Chapter C describes the available scripts provided by xSAP.
- Chapter D describes the available commands.

This document is not self contained, and assumes that the reader is familiar with nuXmv. The interested reader is referred to [20]. A technical description of the engines underlying nuXmv and xSAP can be found in [16, 6].

[1] `http://www.compass-toolset.org`
[2] `http://autogef-project.fbk.eu`
[3] `http://fame-project.fbk.eu`
[4] `http://hasdel-project.fbk.eu`

Chapter 2

Methodological Overview

2.1 A Formal Approach to System Design

We consider a unified process that covers the modeling and the verification of complex and safety critical systems. The process, depicted in Figure 2.1, aims to support the early design phases by developing systems at an architecture level.

Requirements Validation: In order to ensure the quality of requirements, they can be validated independently of the system. This includes both property consistency (i.e., checking that requirements do not exclude each other), property assertion (i.e., checking whether an assertion is a logical consequence of the requirements), and property possibility (i.e., checking whether a possibility is logically compatible with the requirements). Altogether these features allow the designer to explore the strictness and adequacy of the requirements. Expected benefits of this approach include traceability of the requirements and easier sharing between different actors involved in system design and safety assessment. Furthermore, high-quality requirements facilitate incremental system development and assessment, reuse and design change, and they can be useful for product certification.

Functional verification: Analyzing operational correctness is the first step to be performed during the system development lifecycle. It consists in verifying that the system will operate correctly with respect to a set of functional requirements, under the hypothesis of nominal conditions, that is, when software and hardware components are assumed to be fault-free. One particular instance of this general model-checking problem that is specifically supported by the toolset is deadlock checking, i.e., ensuring that the system does not give rise to terminating computations. This is usually required for reactive systems. Moreover the toolset offers the feature to interactively simulate the execution of the system.

Fault Extension: In general the behavior of a critical system is modeled in two parts: nominal and faulty. The former describes the system when it is not affected by any faulty behavior, while the second extends the nominal one with the possibility to have some undesirable behavior (e.g. a battery does not provides electricity, a link becomes broken, etc). The Model Extension permits to keeps disjoint the nominal and the faulty behavior of the system such that is possible to have a greater verification coverage. Fault

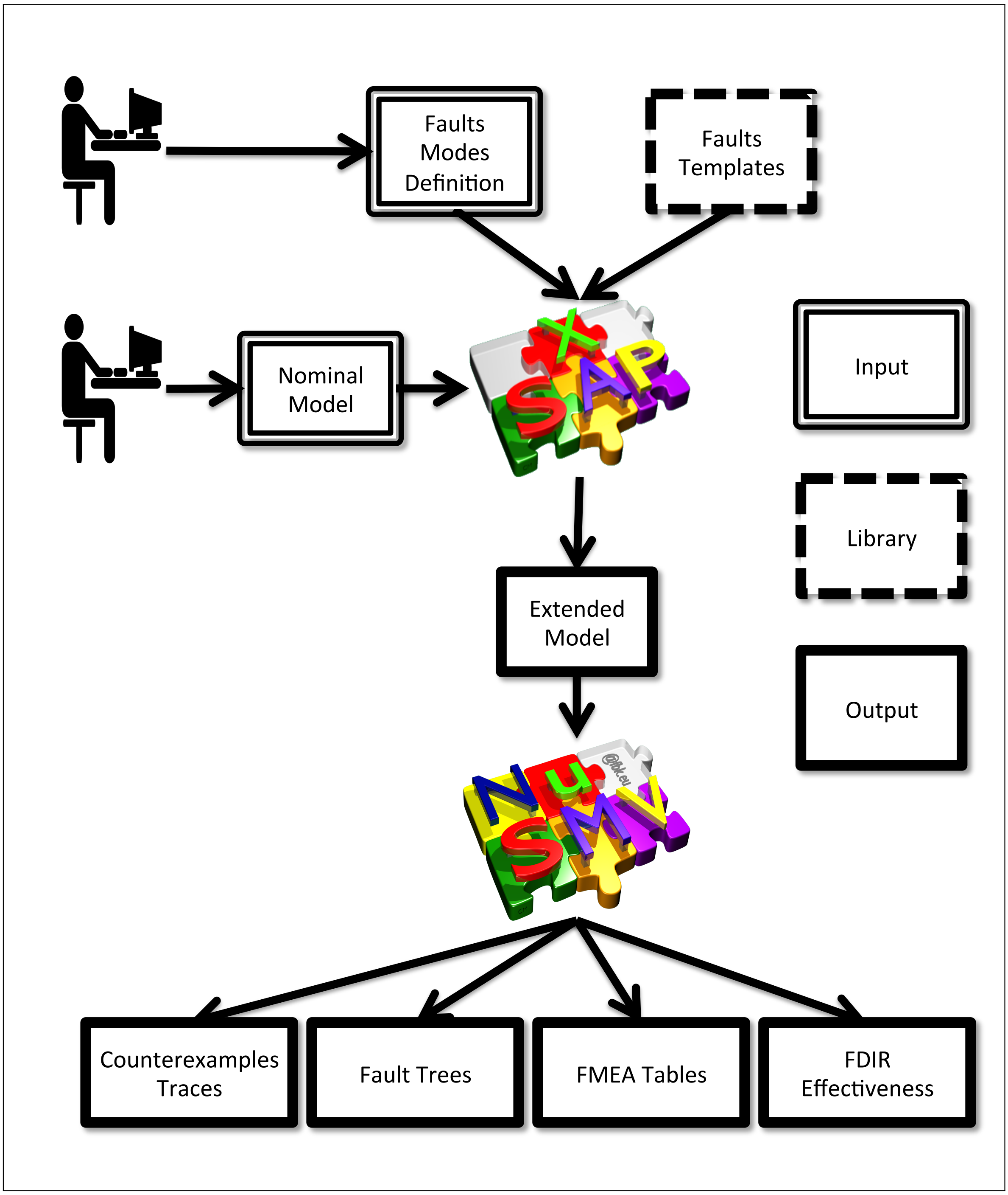


Figure 2.1: Overview of the methodology

extension can be carried out manually (compare Section 3.1) or automatically (compare Section 3.2)[1].

Safety and Dependability Analysis [22, 21, 24, 23]: Analyzing system safety and de-

[1]Figure 2.1 depicts the automated fault extension mechanism, which takes as input fault templates taken from a library, and set of fault extension instructions that define how the faults are to be injected into the nominal model

pendability is a fundamental step that is performed in parallel with system design and verification of functional correctness. The goal is to investigate the behavior of a system in degraded conditions (that is, when some parts of the system are not working properly, due to malfunctions) and to ensure that the system meets the safety requirements that are required for its deployment and use. Key techniques in this area are (dynamic) fault tree analysis, (dynamic) Failure Modes and Effects Analysis (FMEA), fault tolerance evaluation, and criticality analysis.

Fault Detection and Identification Analysis: Fault tolerant systems often implement some mechanisms to detect, identify and recover from, faults – i.e. an FDIR (Fault Detection, Identification and Recovery) sub-system. Fault Detection and Identification (FDI) is carried out by dedicated modules, called FDI components, running in parallel with the system. The detection task is the problem of understanding whether a component has failed, whereas the identification task aims to understand exactly which fault occurred. Typically, faults are not directly observable and their occurrence can only be inferred by observing the effects that they have on the observable parts of the system. FDI components take as input sequences of observations (made available by sensors) and trigger a set of alarms in response to the occurrence of faults.

2.2 Tool support with nuXmv and xSAP

The process is supported by two tools: nuXmv and xSAP. nuXmv mainly targets the phases of requirements analysis, design and verification. xSAP supports the phases of fault extension and safety assessment.

The steps of the methodology covered by functionalities of the nuXmv model checker are "traditional" formal verification steps. nuXmv is an extension of the NuSMV model checker along two key directions: it has a much stronger engine to deal with finite state models, and allows to deal with infinite state transition systems. The nuXmv model checker [15], that can be downloaded at the nuXmv webpage (http://nuxmv.fbk.eu/) More details at [15]. In this section we only present a overview of the nuXmv, and refer the reader to the nuXmv User Manual [20] for details.

xSAP builds upon nuXmv in the following way. In xSAP the nominal models, as well as the models resulting from fault extension, can be expressed in the language of nuXmv, which is, modulo minor variations, the language of NuSMV.

The properties used to represented the behavior of the system, be it nominal or extended, are written in form of temporal properties (either as invariants, LTL, or CTL) in the nuXmv language.

nuXmv provides several functionalities, that are also available in xSAP. These include functional verification, simulation, deadlock checking.

Chapter 3

Fault Extension

Given a model describing the nominal behaviour of a system, *fault extension* is a process that, based on a specification of the possible faults, returns a model whose behaviour takes into account also faulty behaviors. xSAP allows the user to handle the fault extension phase manually or automatically.

Manual fault extension is carried out through the manual definition of a `Fault Modes` file (compare Section 3.1).

On the other hand, the automated fault extension (compare Section 3.2) relies on a set of Fault Extension Instructions (`FEI`), and a library of faults, defining fault effects and fault dynamics (compare Section 3.2.1). The `FEI` instructs how the faults should be injected into the nominal model. As a result of the extension, the nominal model is extended with faulty behaviors and a set of *fault variables* that enable such behaviors.

In a nutshell, in automated fault extension, the set of fault variables is automatically added to the nominal model by the tool, according to the `FEI`. In manual fault extension, instead, the user has to declare a set of existing variables in the (manually created) extended model, to be the set of fault variables. The set of fault variables is used by xSAP to carry out the subsequent analyses – compare Chapter 4.

Once the fault extension phase is complete, it is possible to list the set of fault variables using the shell command `show_fault_variables`, as shown in Appendix D.4.

3.1 Manual Fault Extension

Manual extension of a nominal model is possible through the execution of specific commands available in the tool's shell. The user needs to write a `Fault Modes` file (xml format) which must be specified when invoking the tool (see section D.1).

Once the `Faults Mode` file has been loaded properly, several analyses can be performed, for instance Fault Tree Analysis (see Section 4.2).

3.1.1 XML Format

The `Fault Modes` file used in manual fault extension must be compliant with the xsd provided in file `data/schema/failure-modes.xsd`.

The root element, named `<compass>`, consists of the following elements:

- `<fmlist>`: required list of faults modes

- `<cclist>`: optional list of common causes
- `<obslist>`: optional list of observables[1]

Faults modes can be defined using the tag `<fm>` and are required to have the attributes `name` and `nominal_value`. Optionally, fault modes can be associated to a probability (attribute `probability`) and to a history variable (attribute `history` – see Section 4.1 for additional details). In case we want to define a latent fault mode (compare Section 4.2.1), we need to add a child element named `<latent>` containing the related probability as an attribute.

If common causes are present (compare Section 4.5), they are introduced by element `<cc>` and must have an attribute `name` representing the name and an attribute `fm` representing the fault mode they refer to; in addition, it is possible to specify a proability through the attribute `probability`. Each common cause is associated to at least one mode (identified by the child element `<cc_mode>`) which requires to have specified a `name` and an interval (represented through attributes `low` and `high`) within the failure will be raised.

An example of valid `Fault Modes` file is depicted in figure 3.1.

```xml
<?xml version="1.0"?>
<compass>
  <fmlist>
    <fm name="SC.G3.Gen_StuckOff.mode_is_stuckAt_Off"
        nominal_value="FALSE" probability="0"></fm>
    <fm name="SC.G2.Gen_StuckOff.mode_is_stuckAt_Off"
        nominal_value="FALSE" probability="0"></fm>
    <fm name="SC.G1.Gen_StuckOff.mode_is_stuckAt_Off"
        nominal_value="FALSE" probability="0"></fm>
  </fmlist>
  <cclist>
  </cclist>
  <obslist>
  </obslist>
</compass>
```

Figure 3.1: Example of `Fault Modes` file

3.2 Automated Fault Extension

The automatic extension of a nominal model is possible through the execution of specific commands available in the tool's shell, or through a script which simplifies the whole process. The user needs to write a `FEI` file (human-readable textual and xml formats are both supported), and then the `FEI` file is processed when carrying out the extension.

The fault extension process enriches the model in two directions:

- it associates a faulty behaviour model with an instance of a nominal component implementation;

[1]This tag is present for historical reasons, and not currently used

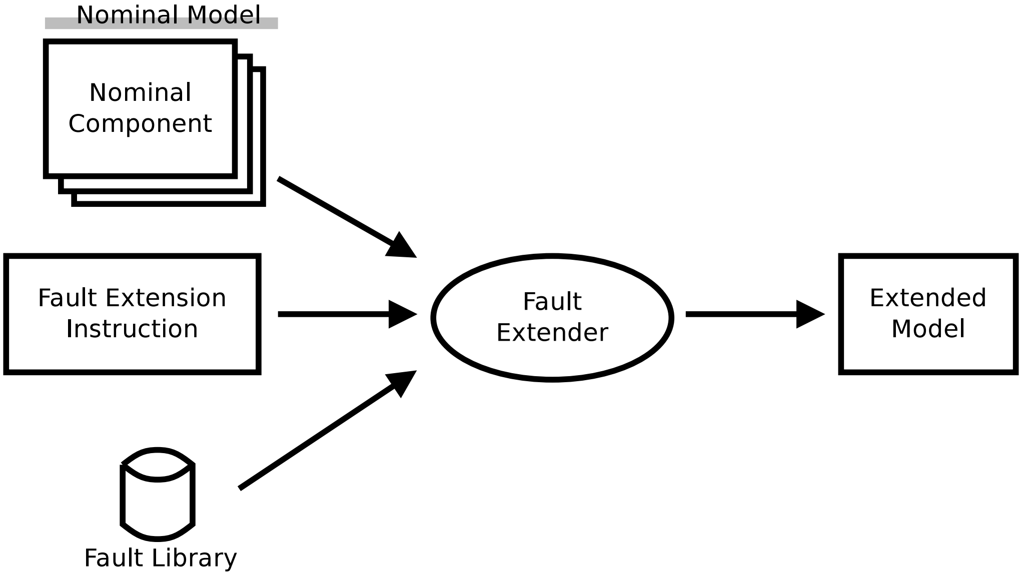


Figure 3.2: Automated Fault Extension flow

- it defines the dynamics controlling the occurrence of a faulty behavior in the respective component.

The process consists in extending a set of `NC`s (nominal components) and passes through the execution of the `Fault Extender` tool, which takes as inputs:

- The set of Nominal Components
- The Fault Library (see 3.6)
- The Fault Extension Instruction (`FEI`) containing a description of the way each `NC` should be extended (see section 3.4).

The `Fault Extender` produces the Extended Model as output, i.e. a `SMV` file containing the Nominal Model automatically extended with the faulty behavior.

The steps of the process are explained in greater details below.

3.2.1 Fault Extension Instructions

The language for fault extension is built on the following concepts.

A FE is a set of Nominal Components (`NC`) each associated with its set of Fault Slices.

Each Fault Slice contains the fault modes and their dynamics which affect a slice of the `NC` variables.

Formally, a FE is a set:

FE := {<NC, { FS } >}
FS := <AS, {<EM, LDM>}, GDM>

where:

`NC` a nominal component,

FS   is a fault slice,

where a FS is composed of

AS   is a non-overlapping subset of the symbols of NC which are affected by the fault

EM   is an Effect Model

LDM   a Local Dynamics Model

GDM   is a Global Dynamics Model

3.2.2 Fault Slice

A Fault Slice (FS) is a set of Fault Modes with associated Global Dynamics Model and Affected Symbols (AS).

Formally:

```
FS := <AS, {<EM, LDM>}, GDM>
```

A FS applies its effects to a subset AS of the set of symbols of a NC. Intuitively, FSs allows for composing Fault Modes which may be thought as basic faults, to build more complex faults.

All Fault Slices of a NC are implicitly in cross-product, each operating independently on its slice over NC's variables.

3.2.3 Fault modes

One fault mode (fm) is made of:

- One Effects Model (EM)
- One Local Dynamics Model (LDM)

One fm has two modalities: nominal and fault.

Effect Model (EM)

Effects are constraints over a set of variables which are intended to be bounded to corresponding affected symbols AS (variable or defines) in the NC, whose values received through the FS interface. Each EM determines what are the effects of entering and during the fault mode.

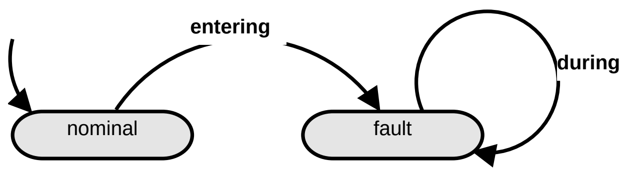


Figure 3.3: States and transitions of an EM

The effect can be function of the internal state of the EM. Predicates is_nominal and is_fault are available in EM to tell which is the current/next mode.

An EM applies its effects on an AS by writing to a corresponding output variable (OV). The EM can read the value of the AS through a corresponding input value. If two or more EMs write to the same AS, all the corresponding OVs must have the same type (which is type of the AS).

Local Dynamics Model (LDM)

LDM models how the fm moves between the nominal and fault modes. One or more fault events can be defined, and transitions between the modalities are defined. Like for EMs, predicates is_nominal and is_fault are available in EM to tell which is the current/next mode.

Interface of a fm

The fm interface results from the combination of the EM and the LDM.

The fm Interface is nominally made of:

- One or more input values, which carry the value of corresponding variables/defines in the NC for binding.
- One or more output variables which are constrained in the EM, each overwriting a corresponding input variable/define (in AS) in the NC passed through a corresponding input value.
- One or more additional read-only input parameters for constants, or other values coming from the NC.
- One or more template parameters which are resolved by string substitution.
- One or more events, defined and used in the EM and LDM.

Instantiation of a fm

One EM and one LDM are taken from the respective generic libraries, and associated. This requires that all template strings are resolved. The fm construction produces a fm which needs to be contextualized.

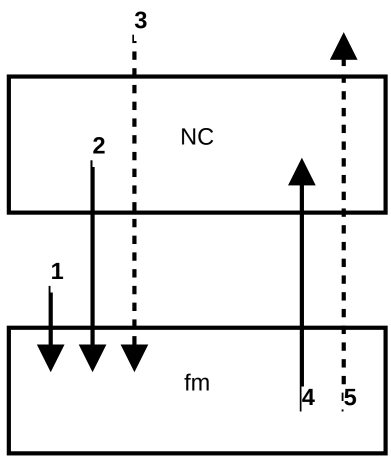


Figure 3.4: Binding NC's content into one fm

The fm interface can take values from constants (1), from the NC's internals (2), and from the NC's interface (3). The fm Interface can output (constrain) data and events to the internal state of NC (4) or to its interface (5).

All fms belonging to a FS are associated with one NC. This requires that:

- All input variables/defines which have to be bounded in the NC are associated with the corresponding output variables of the fm.
- All events of the fm which need to be constrained by the NC are connected to the corresponding values in the NC (NC's parameters and local symbols).
- All other input values are resolved, either attached to a constant value, or to values in the NC (NC's parameters and local symbols).
- All remaining template strings are resolved.

3.2.4 Global Dynamics Model

GDM defines the dynamics of composition of several fm's of each FS.

Two or more instantiated fms can be composed to define more complex behaviours.

1. A set of instantiated fms are chosen.
2. Transitions among them are defined.
3. New events may be added at this stage to be used as triggers in transitions.

In a FS's GDM the nominal modes of all instantiated fms are collected in one unique nominal mode, to compose a daisy-shaped automaton, whose leaves are the fail modes.

In Figure 3.5, three fms are composed, each has its LDM (in gray), but in addition also transitions between fail2 and fail1, and between fail2 and fail3 are defined (in blue). Those transitions can be labeled either with:

- Events which exist in the *target* fm, or
- *new* events created at this stage.

In the example e1 is an event occurring in the fm fail1. new_ev is instead a newly created event.

3.3 Common Cause

A fault extension specification is made of one or more module's extensions, and optionally by one or more Common Causes (CCs).

Each CC represents a set of failures, possibly occurring in different components and different times, for which the assumption of fault independence does not hold. For example, failure of one power generator may happen standalone, or may happen as a consequence of a CC. In the latter case, the CC may specify that two generators may fail, and that the two failures are

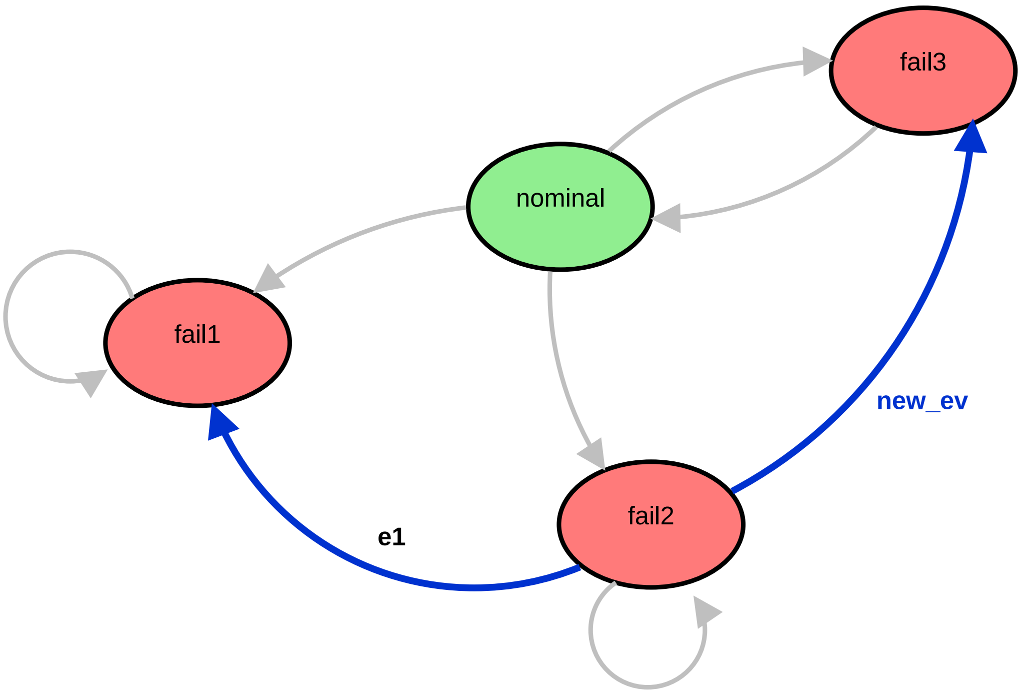


Figure 3.5: Global Dynamics Model among three `fm`s

dependent, e.g., the first generators fails, and then the second generators fails between 1 and 3 discrete time-steps *as a consequence* of the common cause.

Each `CC` specifies an initiating event (the *common cause event*, which is distinct from the component `fm` themselves) and the set of the component `fm` with the corresponding timings (specified as intervals). Each interval specifies the time bounds (expressed relative to the time of the initiating event) at which the `fm` can take place.

3.4 The Fault Extension Instruction language

`FEI` is the way a user specifies how a Nominal Model has to be extended with faulty behaviours.

`FEI` can be specified either with a XML specification, or in a human-readable textual format.

The EBNF-like grammar of the `FEI` is presented below. A complete example of usage of `FEI` is presented in Section 7.3.2.

```

<dot> ::= '.'
<colon> ::= ':'
<semi-colon> ::= ';'
<comma> ::= ','
<lpar> ::= '('
<rpar> ::= ')'
<lbra> ::= '{'
<rbra> ::= '}'
<rw-op-lt> ::= "<<"
<rw-op-gt> ::= ">>"
<eq-op> ::= '='
<range-op> :== ".."
<trans-begin> ::= "-["
<trans-end> ::= "]->"
```

```

<line-comment> ::= -- .* \n
<multiline-comment> ::= /-- [.\n]* --/

<id> ::= [A-Za-z_][A-Za-z_0-9-]*
<wildcard-id> ::= [A-Za-z_*\[\]?][A-Za-z_0-9*\[\]?-]*

<full-id> ::= <id> | <id> <dot> <full-id>
<wildcard-full-id> ::= <wildcard-id> | <wildcard-id> <dot> <wildcard-full-id>

<digits> ::= [0-9]+
<number> ::= (+|-)? <digits>

<float-number> ::= <number> <dot> (<digits>)?
<exp-number> ::= <float-number> (e|E) <number>
<real-number> ::= <float-number>
                | <exp-number>

# ---------------------------------------------------------------------------
<start> ::= <fault-extension>


# ---------------------------------------------------------------------------
<fault-extension>
    FAULT EXTENSION <id>
      (<mod-extension>)+
      (<common-causes>)?


# ---------------------------------------------------------------------------
# Fault slices and modes from here
# ---------------------------------------------------------------------------
<mod-extension> ::=
    EXTENSION OF MODULE <full-id>
      (<instances>)?
      <slice>+

<slice> ::=
    SLICE <slice-id> (<instances>)? AFFECTS <var-list> WITH
       <fault-mode>+
       <global-dynamics>?

<instances> ::=
    FOR INSTANCES <wildcard-full-id-list>

<fault-mode> ::=
    MODE <mode-id> (probability-value-mode)? <colon>
         <local-dynamics-model-id> <effect> <semi-colon>

<global-dynamics> ::=
    GLOBAL DYNAMICS
    <new-event>*
    <trans>+

<new-event> ::=
           <event-par> <semi-colon>

<slice-id> ::= <id>
<mode-id> ::= <id>

<local-dynamics-model-id> ::= <id>

<effect> ::=
   <effect-id> <lpar> <par-list> <rpar>

<effect-id> ::=
   <id>

<trans> ::=
        TRANS <trans-mode-id>
```

```
            <trans-begin> <full_id>? <trans-guard>? <trans-end>
            <trans-mode-id> <semi-colon>

<trans-guard> ::=
         when <simple-expression>

<trans-mode-id> ::=
          | <mode_id>.nominal
          | <mode_id>.fault
          | nominal

<var-list> ::=
            <id>
            | <id> <comma> <var-list>

<par-list> ::=
            <par>
            | <par> <comma> <par-list>

<par> ::=
      <data-par>
      | <event-par>
      | <template-par>

<data-par> ::=
      data <id> <rw-op-expr>?

<event-par> ::=
      event <id> <rw-op-expr>?

<rw-op-expr> ::=
    | <rw-op-lt> <simple-expression>
    | <rw-op-gt> <id>

<template-par> ::=
      template <id> <eq-op> <id>

<wildcard-full-id-list> ::=
      <wildcard-full-id>
      | <wildcard-full-id> <comma> <wildcard-full-id-list>


# ------------------------------------------------------------------------
# Common cause from here
# ------------------------------------------------------------------------
<common-causes> ::=
    COMMON CAUSES
      (<common-cause>)*

<common-cause> ::=
    CAUSE <id> (probability-value-cc)?
    (<cc-module-modes>)+

<cc-module-modes> ::=
    MODULE <full-id>
      (<instances>)?
      (<cc-mode>)+

<cc-mode> ::=
    MODE <slice-id><dot><mode-id> <cc-range> <semi-colon>

<cc-range> ::=
    WITHIN <digits> <range-op> <digits>

<probability-value-mode> ::=
    <lbra> <real-number> (, latent:yes|no, latent_prob:<real-number>)?<rbra>
    | <lbra> prob:<real-number> (, latent:yes|no, latent_prob:<real-number>)?<rbra>

<probability-value-cc> ::=
    <lbra> <real-number> <rbra>
```

```
    | <lbra> prob:<real-number> <rbra>
```

3.5 FEI semantics

3.5.1 Common Causes

The specification of `CC` is made of a set of Cause specifications. Each Cause is made of a set of fault modes each associated with a (discrete) time interval. The set of fault modes is partitioned wrt the modules they belong to.

In the example, the `FEI` contains a single Cause `CC1`, involving *all* instances of module `Switch`, and different fault modes dynamics for the `Generator` s.

```
COMMON CAUSES
  CAUSE CC1
       MODULE Generator
         FOR INSTANCES SC.G1
         MODE Gen_StuckOff.stuckAt_Off WITHIN 0 .. 0;

       MODULE Generator
         FOR INSTANCES SC.G[23]
         MODE Gen_StuckOff.stuckAt_Off WITHIN 1 .. 3;

       MODULE Switch
         MODE Switch_StuckClosed_StuckOpen.stuckAt_Open WITHIN 2 .. 3;
         MODE Switch_StuckClosed_StuckOpen.stuckAt_Closed WITHIN 3 .. 4;
```

In the example, when fault `CC1` occurs, `SC.G1` immediately may stuck off, which lead to having both `SC.G2` and `SC.G3` fail stuck off after 1 to 3 steps. All `Switch` are also involved, as they may stuck to/from stuck open and stuck closed. The example is certainly a bit artificial, but the intention is use the example to informally present the semantics underlying the `CC`.

Each `MODULE` M specification contains a set of references to fault modes which were defined in the extension of module M. Each fault mode reference is associated with a (discrete) time interval which is the delay of the occurrence of that fault mode after the Common Cause happen. For example, fault `SC.G1.Gen_StuckOff.stuckAt_Off` may happen instantaneously (0 steps) after fault `CC1` happen, while fault `Gen_StuckOff.stuckAt_Off` for instances `SC.G2` and `SC.G3` may happen non-deterministically 1 to 3 discrete steps after `CC1`.

Occurrence of a Fault Mode A fault mode *may happen* here means that after the `CC` happen, a fault mode may be not able to happen due to the history and modeling reasons. For example, suppose a valve may fail by exploding or stucking at closed, and suppose the `GDM` does not allow for switching from "exploded" mode to "stuck-at-closed" mode. Now a `CC` involving an exploded valve may still happen, although the fault mode "stuck-at-closed" will never happen as a consequence of the `CC`.

Multiple instances Each `MODULE` specification identifies a set of instances (all if instances are not explicitly identified). Notice that the set of instances of `MODULE` M is a subset of the set of extended instances of module M. As for the fault extension, the `FOR INSTANCES` specification contains a list of patterns (wildcard) to filter existing names. Empty sets are not allowed.

A module instance m of module M can occur only once in each Common Cause. In the example `SC.G1` involved in `MODULE` at line 3, and cannot occur in any other `MODULE` of Cause `CC1`.

When Cause's `MODULE` specification involves more than one instances, the intended semantics is that the `CC` involves *all* specified instances at the same time. For example, `MODULE` specification at line 7 involves both `SC.G2` and `SC.G3`. When at time t `CC1` happen, in a time $t + \Delta$ with $1 \leq \Delta \leq 3$, $\Delta \in \mathbb{N}$, *all* instances `SC.G2` and `SC.G3` which can enter fault mode `Gen_StuckOff.stuckAt_Off` will fail with that fault mode [2].

Multiple Fault Modes As mentioned, Cause's `MODULE` specification allows for multiple fault modes references. This is the set of fault modes (and their timings) which may follow the occurrence of the `CC`. Since all fault modes in a `MODULE` specification affect the same instances, constraints are automatically added during the extension to avoid that two or more incompatible fault modes are activated at the same time. For example, fault modes at lines 12 and 13 may happen for all `Switch` instances 2 to 3 steps after `CC1` occurs [3]. The automatically added constraints avoid a deadlock by constraining the modes not to occur at the same time step.

3.6 The Faults Library

Name	**Parameters**[4]	**Transitions**	
		Entering	**During**
StuckAtByReference_I	term	next(varout) = next(term)	next(varout) = next(term)
StuckAtByReference_D[5]	term	next(varout) = term	next(varout) = term
StuckAtByValue_I	term	next(varout) = next(term)	next(varout) = varout
StuckAtByValue_D	term	next(varout) = term	next(varout) = varout
Frozen	—	next(varout) = input	next(varout) = varout
NonDeterminismByReference_Num_I	min_bound, max_bound	next(varout) >= next(min_bound) & next(varout) <= next(max_bound)	next(varout) >= next(min_bound) & next(varout) <= next(max_bound)
NonDeterminismByReference_Num_D	min_bound, max_bound	next(varout) >= min_bound & next(varout) <= max_bound	next(varout) >= min_bound & next(varout) <= max_bound
NonDeterminismByValue_Num_I	min_bound, max_bound	next(varout) >= next(min_bound) & next(varout) <= next(max_bound)	next(varout) = varout
NonDeterminismByValue_Num_D[6]	min_bound, max_bound	next(varout) >= min_bound & next(varout) <= max_bound	next(varout) = varout
NonDeterminismByReference_Bool	—	next(varout) = FALSE | next(varout) = TRUE	next(varout) = FALSE | next(varout) = TRUE
NonDeterminismByValue_Bool	—	next(varout) = FALSE | next(varout) = TRUE	next(varout) = varout
Conditional_I	condition, then_term, else_term	next(varout) = (CONDITION_AT_ENTRANCE ? next(then_term) : next(else_term))	next(varout) = (CONDITION_AT_ENTRANCE ? next(then_term) : next(else_term))
Conditional_D	condition, then_term, else_term	next(varout) = (CONDITION_AT_ENTRANCE ? then_term : else_term)	next(varout) = (CONDITION_AT_ENTRANCE ? then_term : else_term)
ConditionalDualOutputs_I	condition, then_term_1, else_term_1, then_term_2, else_term_2	next(varout_1) = (CONDITION_AT_ENTRANCE ? next(then_term_1) : next(else_term_1)) & next(varout_2) = (	next(varout_1) = (CONDITION_AT_ENTRANCE ? next(then_term_1) : next(else_term_1)) & next(varout_2) = (

[2] Assuming there are no other Causes specified for the same instances

[3] Assuming the `GDM` allows for switching between modes `stuckAt_Open` and `stuckAt_Closed`, in one or both directions

		CONDITION_AT_ENTRANCE ? next(then_term_2) : next(else_term_2))	CONDITION_AT_ENTRANCE ? next(then_term_2) : next(else_term_2))
ConditionalDualOutputs_D	condition, then_term_1, else_term_1, then_term_2, else_term_2	next(varout_1) = (CONDITION_AT_ENTRANCE ? then_term_1 : else_term_1) & next(varout_2) = (CONDITION_AT_ENTRANCE ? then_term_2 : else_term_2)	next(varout_1) = (CONDITION_AT_ENTRANCE ? then_term_1 : else_term_1) & next(varout_2) = (CONDITION_AT_ENTRANCE ? then_term_2 : else_term_2)

RampDown	decr
	end_value
		next(varout) = input	next(varout) = case
			ramp_mode = RAMPING_DOWN : varout - decr;
			ramp_mode = RAMPING_DONE : varout;
			esac;

Inverted	—	next(varout) = !input	next(varout) = varout
StuckAtFixed	—	next(varout) = varout	next(varout) = varout
RandomByReference	—	TRUE	TRUE
RandomByValue	—	TRUE	next(varout) = varout
ErroneousByReference	—	next(varout) != next(input)	next(varout) != next(input)
ErroneousByValue	—	next(varout) != next(input)	next(varout) = varout
DeltaOutByReference	delta	next(varout) < (next(input) − delta) | next(varout) > (next(input) + delta)	next(varout) < (next(input) − delta) | next(varout) > (next(input) + delta)
DeltaOutByValue	delta	next(varout) < (next(input) − delta) | next(varout) > (next(input) + delta)	next(varout) = varout
DeltaInRandomByReference	delta	next(varout) >= (next(input) − delta) & next(varout) <= (next(input) + delta)	next(varout) >= (next(input) − delta) & next(varout) <= (next(input) + delta)
DeltaInRandomByValue	delta	next(varout) >= (next(input) − delta) & next(varout) <= (next(input) + delta)	next(varout) = varout
DeltaInErroneousByReference	delta	next(varout) >= (next(input) − delta) & next(varout) <= (next(input) + delta) & next(varout) != next(input)	next(varout) >= (next(input) − delta) & next(varout) <= (next(input) + delta) & next(varout) != next(input)
DeltaInErroneousByValue	delta	next(varout) >= (next(input) − delta) & next(varout) <= (next(input) + delta) & next(varout) != next(input)	next(varout) = varout

Table 3.1: Effect Modes

Name	**Parameters**	**Transitions**		
		Entering	**During**	**Back**
Permanent	—	T: failure	—	—
Transient	—	T: failure	G: !self_fix	T: self_fix
SelfFixWithCounter	counter_max	T: failure	G: counter < counter_max	G: counter >= counter_max

Table 3.2: Local Dynamics

The Fault Library contains the models split for the effects and the local dynamics. The library is made of xml files and optionally correlated with `SMV` file to model the behaviours.

[4]Other than "varout" and "input"
[5]Formerly StuckAt
[6]Formerly NonDeterminism

3.6.1 Effect Model

Stuck-At By Reference

An example of an `EM` which models the effects of stucking-at a given term by reference[7].

```
<effects_model name="StuckAtByReference_D">
  <values>
    <input reads="term" type="Any"
     desc="The value at which the output has to be stuck. Can be a constant or a variable."/>
    <output writes="varout" reads="input"
     desc="The output variable name that reads on the input one"/>
  </values>
  <effect>
    <entering type="smv" local="false">entering.smv</entering>
    <during type="smv" local="false">during.smv</during>
  </effect>
  <raw/>
</effects_model>
```

Notice that `SMV` code can be either inlined in the library xml specification, or can be imported from external files. In the example both `entering` and `during` specification refer external files.

The `entering` specification:

```
-- file: entering.smv
--  varout: the output variable
--  term: The value at which the output has to be stuck. Can be a constant or a variable

next(varout) = term
```

The `during` specification:

```
-- file: during.smv
--  varout: the output variable

next(varout) = term
```

There are two different types of Stuck At By Reference fault mode:

- StuckAtByReference_I for modules with instantaneous reaction (Figure 3.6)
- StuckAtByReference_D for modules with delayed reaction (Figure 3.7)

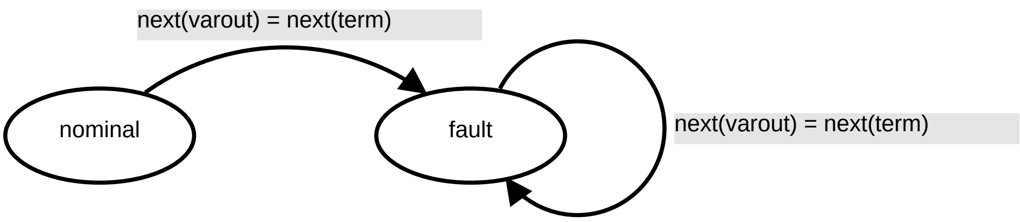


Figure 3.6: `EM` for a stuck-at instantaneous by reference fault mode

[7] By reference means that the value of varout will depend on the values of the parameters in real time

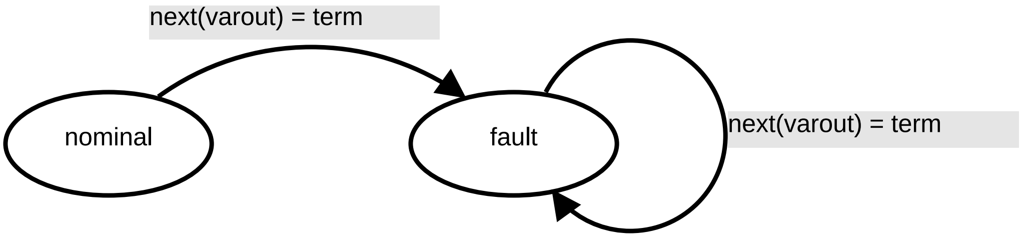


Figure 3.7: EM for a stuck-at delayed by reference fault mode

Stuck-At By Value

EM which models the effects of stucking-at a given term by value[8].

There is two different types of Stuck At By Value fault mode:

- StuckAtByValue_I for modules with instantaneous reaction (Figure 3.8)
- StuckAtByValue_D for modules with delayed reaction (Figure 3.9)

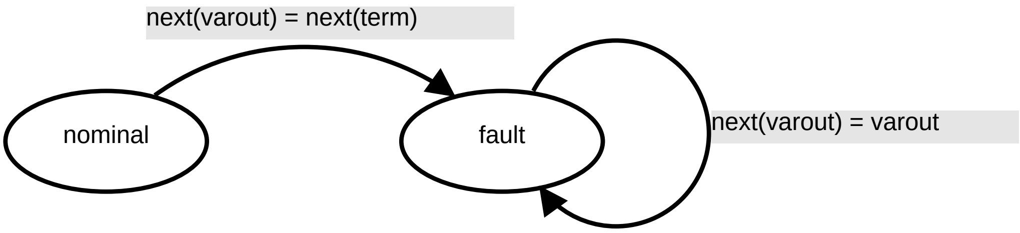


Figure 3.8: EM for a stuck-at instantaneous by value fault mode

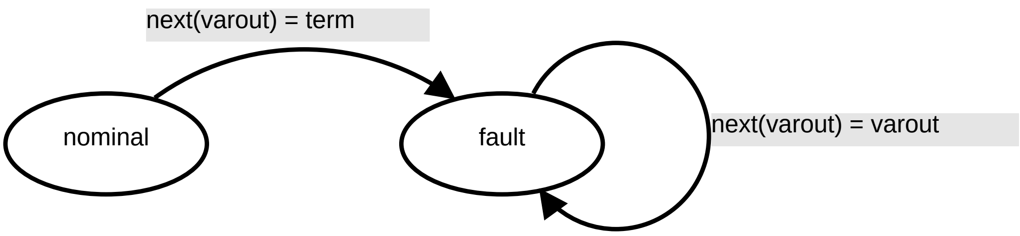


Figure 3.9: EM for a stuck-at delayed by value fault mode

Frozen

EM which models the effects of frozen to the last value.

Non Determinism By Reference for numeric variables

EM which models the effects of giving a random value between a min term and a max term by reference for numeric variables.

There is two different types of Non Determinism By Reference fault mode for numeric variables:

[8] By value means that the value of varout will be evaluated at the entrance in the fault mode, depending on the parameters, and remains the same for all the duration of the fault

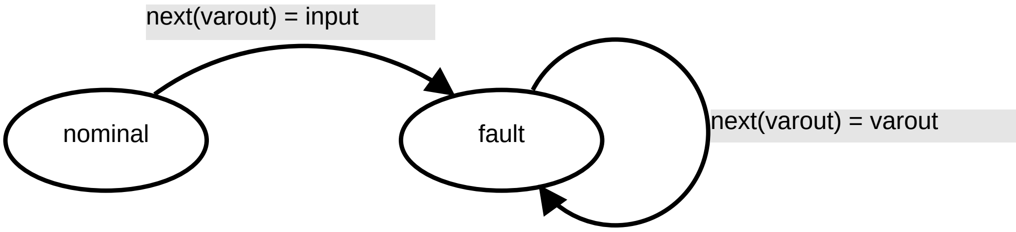


Figure 3.10: `EM` for a frozen fault mode

- NonDeterminismByReference_Num_I for modules with instantaneous reaction (Figure 3.11)
- NonDeterminismByReference_Num__D for modules with delayed reaction (Figure 3.12)

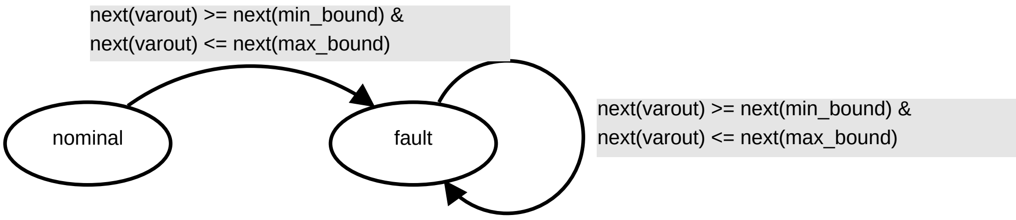


Figure 3.11: `EM` for a non determinism instantaneous by reference fault mode for numeric variables

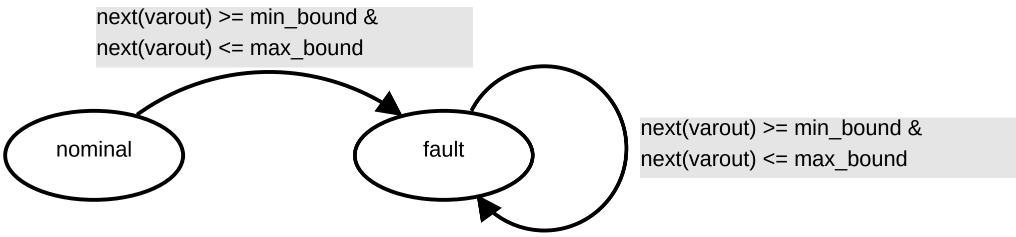


Figure 3.12: `EM` for a non determinism delayed by reference fault mode for numeric variables

Non Determinism By Value for numeric variables

`EM` which models the effects of giving a random value between a min term and a max term by value for numeric variables.

There is two different types of Non Determinism By Value fault mode for numeric variables:

- NonDeterminismByValue_Num_I for modules with instantaneous reaction (Figure 3.13)
- NonDeterminismByValue_Num__D for modules with delayed reaction (Figure 3.14)

Non Determinism By Reference for boolean variables

`EM` which models the effects of giving a random value by reference for boolean variables.

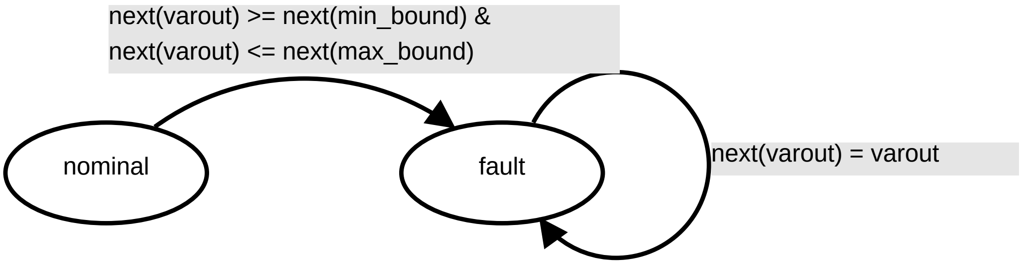


Figure 3.13: `EM` for a non determinism instantaneous by value fault mode for numeric variables

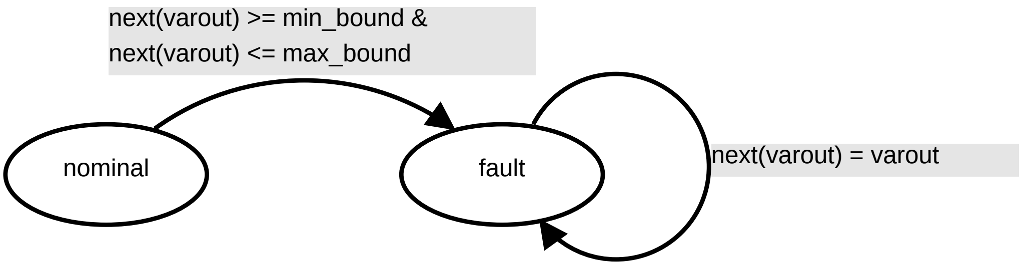


Figure 3.14: `EM` for a non determinism delayed by value fault mode for numeric variables

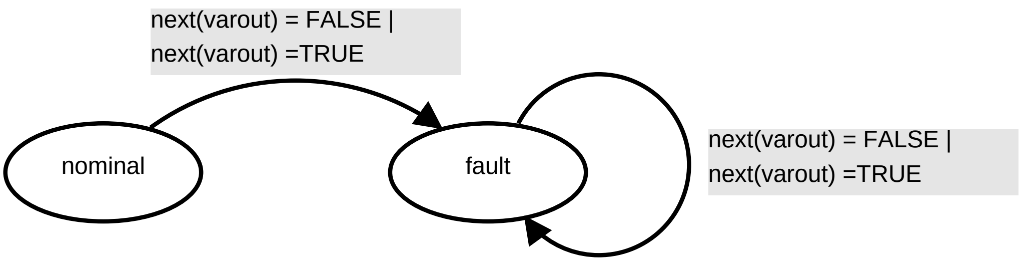


Figure 3.15: `EM` for a non determinism by reference fault mode for boolean variables

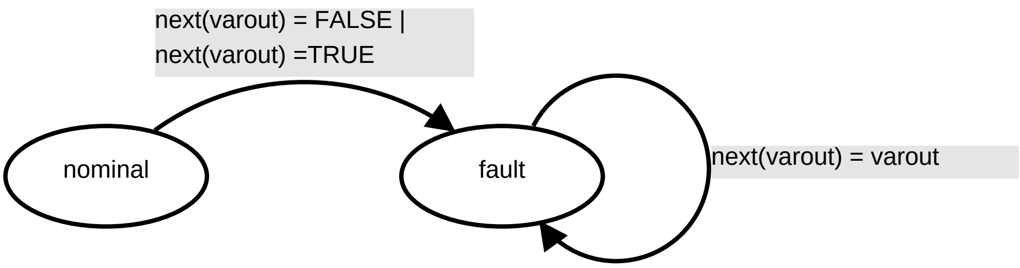


Figure 3.16: `EM` for a non determinism by value fault mode for boolean variables

Non Determinism By Value for boolean variables

`EM` which models the effects of giving a random value by value for boolean variables.

Ramp Down

An example of an `EM` which model the effects of ramping down a symbol.

```
<effects_model name="RampDown" desc="Bla bla">
  <values>
    <input name="decr" type="Integer"/>
    <input name="end_value" type="Integer"/>
```

```
    <output name="varout" reads="input"/>
  </values>
  <effect>
    <entering>next(varout) = input</entering>
    <during>
      next(varout) = case
      ramp_mode = RAMPING_DOWN : varout - decr;
      ramp_mode = RAMPING_DONE : varout;
      esac
    </during>
  </effect>
  <raw>
    VAR ramp_mode : { RAMPING_DOWN, RAMPING_DONE };
    ASSIGN
    init(ramp_mode) := RAMPING_DOWN;
    next(ramp_mode) := case
       !is_fault: RAMPING_DOWN;
       is_fault & varout - decr > end_value: RAMPING_DOWN;
       TRUE: RAMPING_DONE;
     esac;
  </raw>
</effects_model>
```

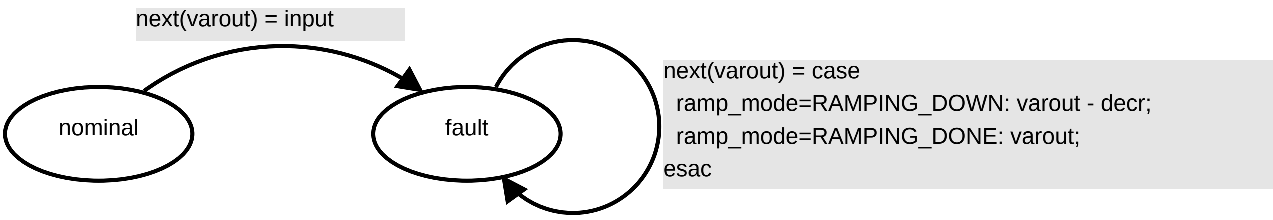


Figure 3.17: `EM` for a ramp-down fault mode

Notice keyword `is_fault` which is a predicate whose truth value is true iff the mode is `fault`. Conversely, predicate `is_nominal` which is true iff the mode is `nominal`.

Furthermore, in general the negation of `is_fault` is not equivalent to `is_nominal`, as when composed with other `fms`, both `is_fault` and `is_nominal` of a single `EM` can be false.

Conditional

An example of an `EM` which model the effects of choosing the varout value based on a boolean condition at the entrance.

```
<effects_model name="Conditional_D">
  <values>
    <input reads="condition" type="Boolean"/>
    <input reads="then_term" type="Any" />
    <input reads="else_term" type="Any" />
    <output writes="varout" reads="input" />
  </values>
  <effect>
    <entering type="smv" local="false">entering.smv</entering>
    <during type="smv" local="false">during.smv</during>
  </effect>
  <raw>
      VAR CONDITION_AT_ENTRANCE : boolean;
      ASSIGN
      init(CONDITION_AT_ENTRANCE) := condition;
      next(CONDITION_AT_ENTRANCE) := case
        !is_fault & next(is_fault) : CONDITION_AT_ENTRANCE;
        is_fault & next(is_fault): CONDITION_AT_ENTRANCE;
       TRUE : next(condition);
      esac;
```

```
  </raw>
</effects_model>
```

There is two different types of Conditonal fault mode:

- Conditional_I for modules with instantaneous reaction (Figure 3.18)
- Conditional_D for modules with delayed reaction (Figure 3.19)

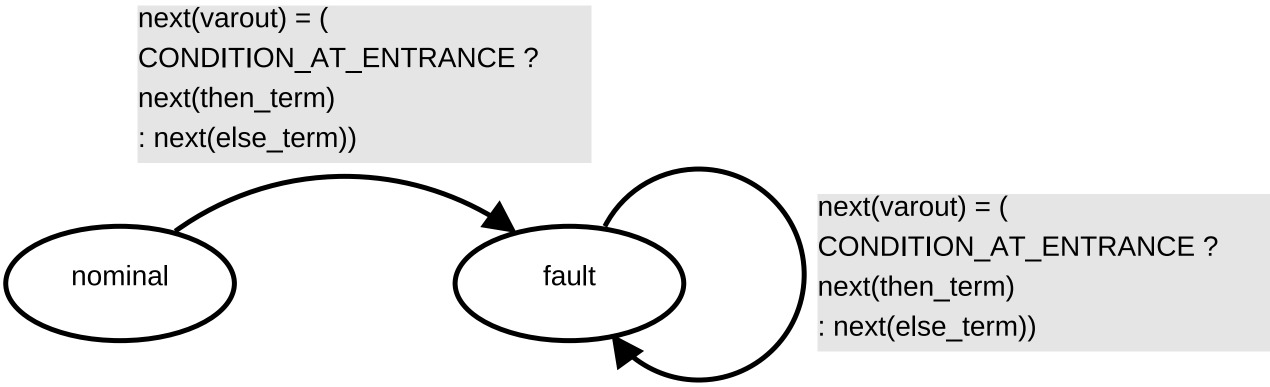


Figure 3.18: EM for a conditional instantaneous fault mode

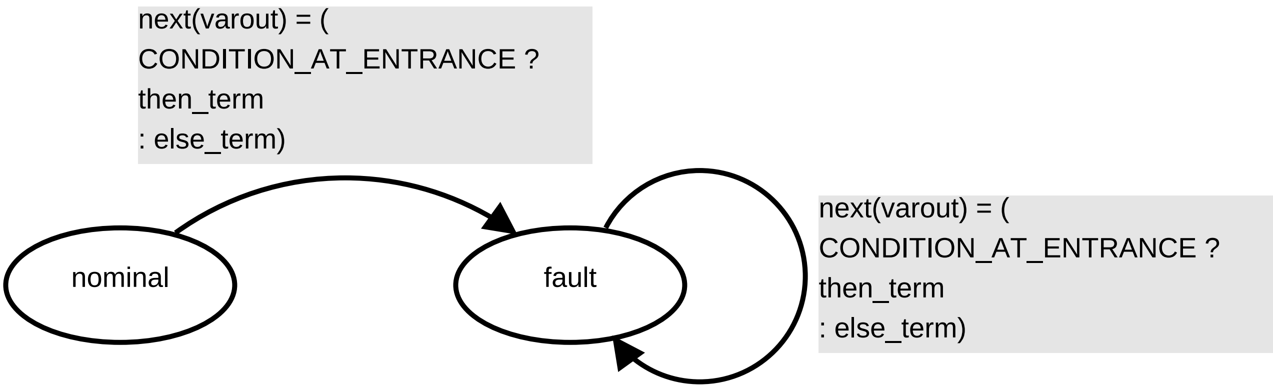


Figure 3.19: EM for a conditional delayed fault mode

ConditionalDualOutputs

An example of an EM which affect two outputs of the NC. Their values will be based on a boolean condition at the entrance.

```
<effects_model name="ConditionalDualOutputs_D">
  <values>
    <input reads="condition" type="Boolean"/>
    <input reads="then_term_1" type="Any" />
    <input reads="else_term_1" type="Any"/>
    <input reads="then_term_2" type="Any" />
    <input reads="else_term_2" type="Any" />
    <output writes="varout_1" reads="input_1" />
    <output writes="varout_2" reads="input_2" />
  </values>
  <effect>
```

```
    <entering type="smv" local="false">entering.smv</entering>
    <during type="smv" local="false">during.smv</during>
  </effect>
  <raw>
      VAR CONDITION_AT_ENTRANCE : boolean;
      ASSIGN
      init(CONDITION_AT_ENTRANCE) := condition;
      next(CONDITION_AT_ENTRANCE) := case
        !is_fault & next(is_fault) : CONDITION_AT_ENTRANCE;
        is_fault & next(is_fault): CONDITION_AT_ENTRANCE;
        TRUE : next(condition);
      esac;
  </raw>
</effects_model>
```

There is two different types of ConditonalDualOutputs fault mode:

- ConditionalDualOutputs_I for modules with instantaneous reaction (Figure 3.20)
- ConditionalDualOutputs_D for modules with delayed reaction (Figure 3.21)

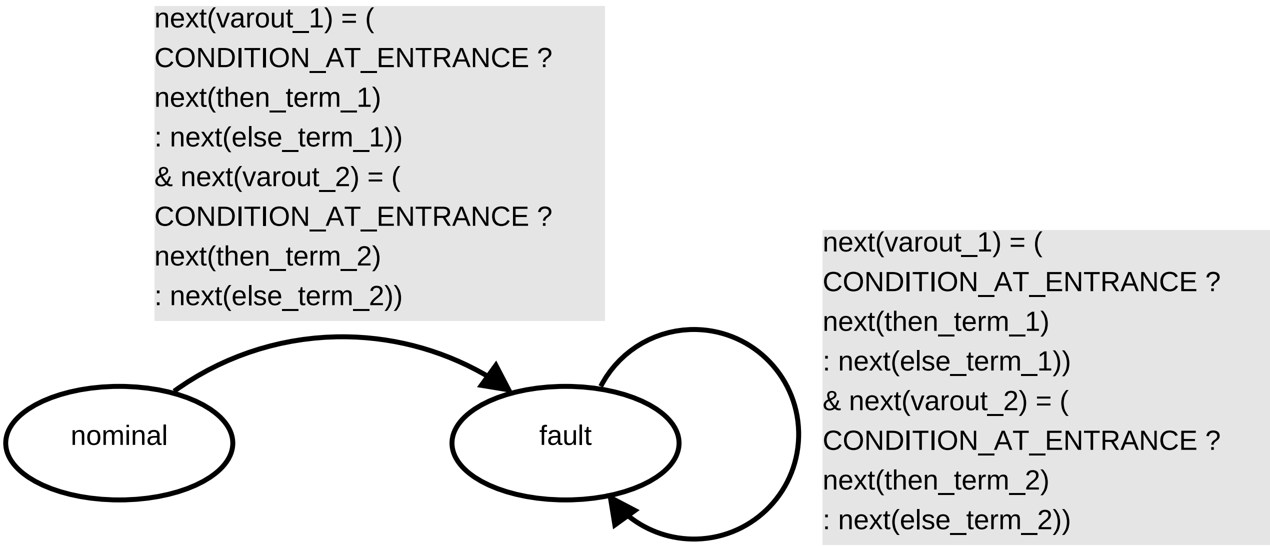


Figure 3.20: `EM` for a conditional instantaneous fault mode for two outputs

Inverted

`EM` which models the effects of being stuck at the inverted last value.

Stuck At Fixed

`EM` which models the effects of being stuck at a fixed random value. The differences with Non Determinism fault modes is that here the value to apply is randomly chosen at the beginning of the execution and never changed after.

Random by reference

`EM` which models the effects of giving a random value by reference.

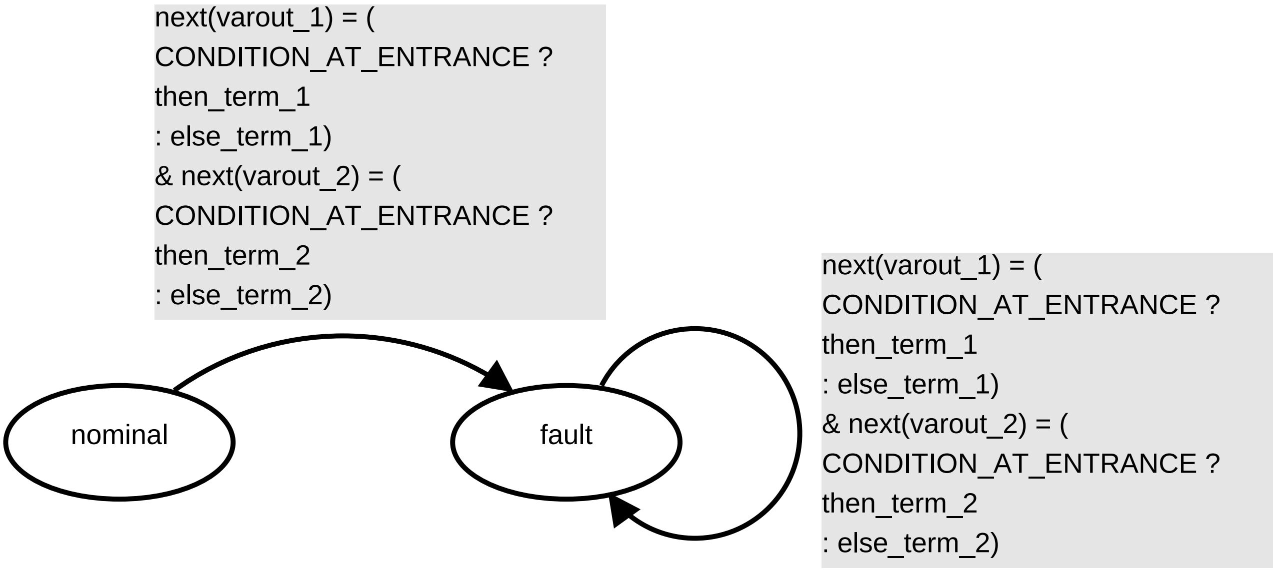


Figure 3.21: EM for a conditional delayed fault mode for two outputs

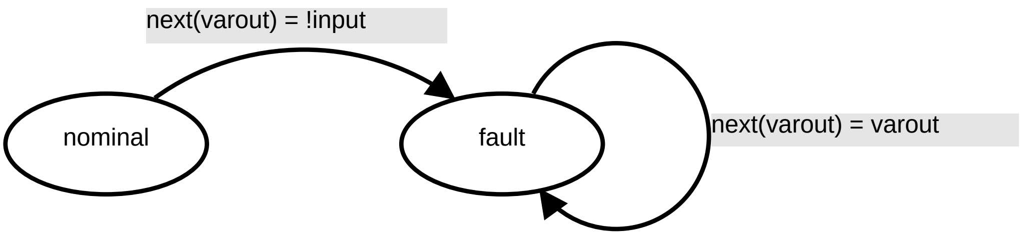


Figure 3.22: EM for an inverted fault mode

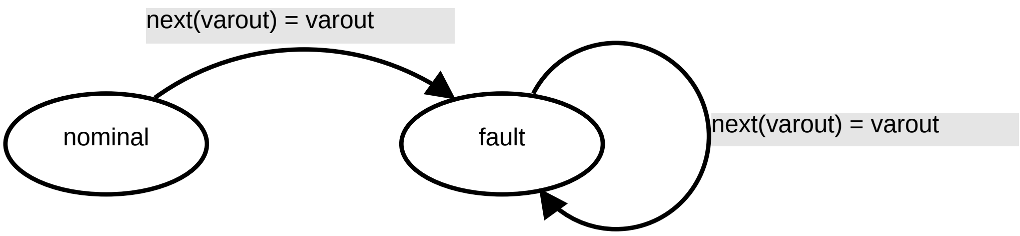


Figure 3.23: EM for an stuck at fixed fault mode

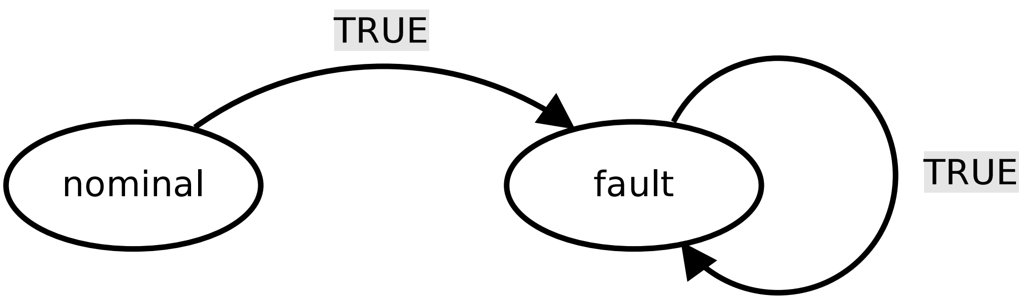


Figure 3.24: EM for a random by reference fault mode

Random by value

EM which models the effects of giving a random value by value.

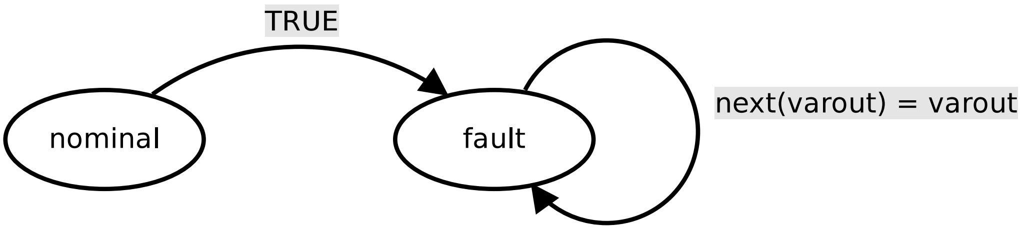


Figure 3.25: EM for a random by value fault mode

Erroneous by reference

EM which models the effects of giving an erroneous value by reference.

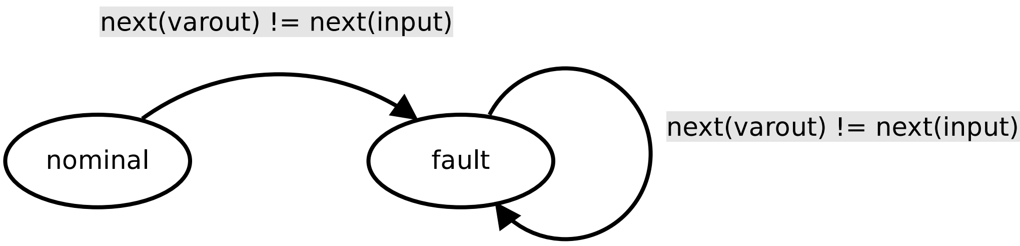


Figure 3.26: EM for an erroneous by reference fault mode

Erroneous by value

EM which models the effects of giving an erroneous value by value.

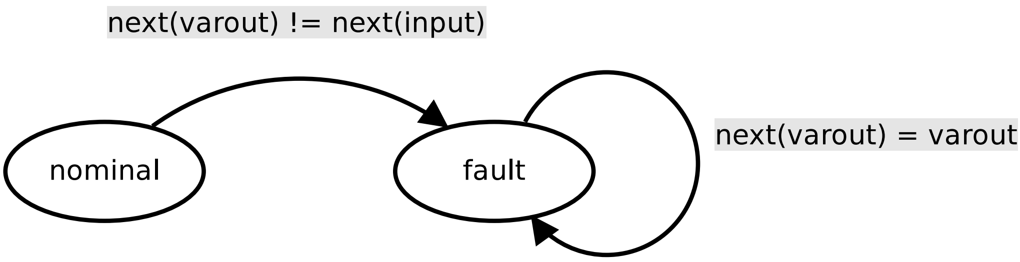


Figure 3.27: EM for an erroneous by value fault mode

Delta out by reference

EM which models the effects of giving a random value out of a delta range from the nominal value, by reference.

Delta out by value

EM which models the effects of giving a random value out of a delta range from the nominal value, by value.

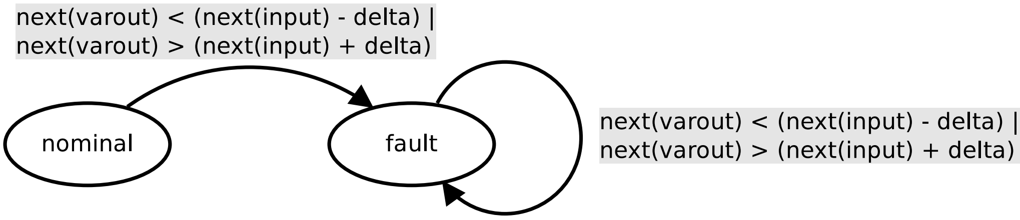


Figure 3.28: EM for a delta out by reference fault mode

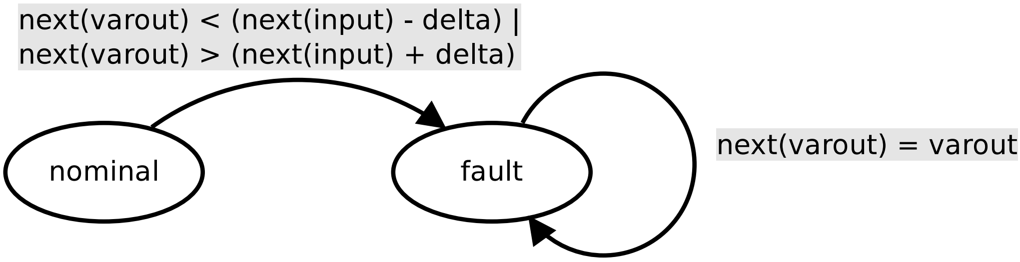


Figure 3.29: EM for a delta out by value fault mode

Delta in random by reference

EM which models the effects of giving a random value in a delta range from the nominal value, by reference.

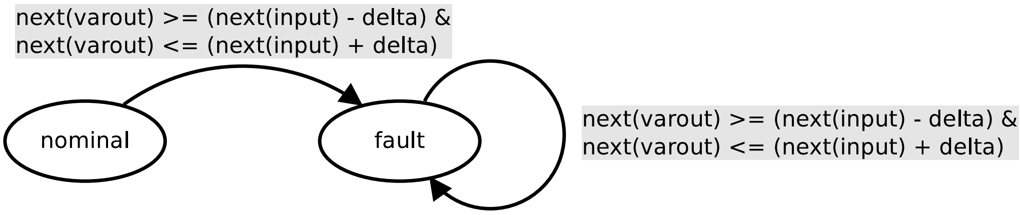


Figure 3.30: EM for a delta in random by reference fault mode

Delta in random by value

EM which models the effects of giving a random value in a delta range from the nominal value, by value.

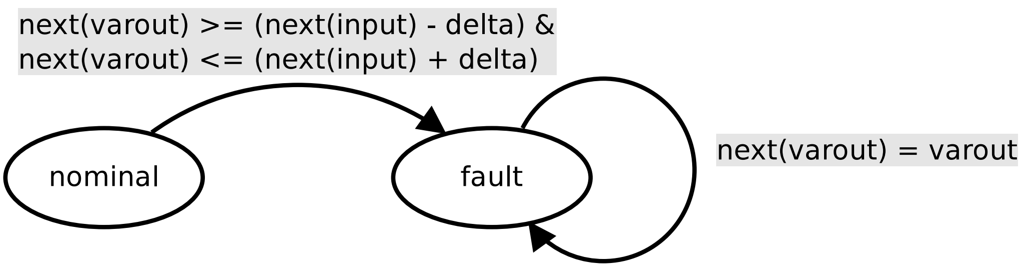


Figure 3.31: EM for a delta in random by value fault mode

Delta in erroneous by reference

EM which models the effects of giving an erroneous value in a delta range from the nominal value, by reference.

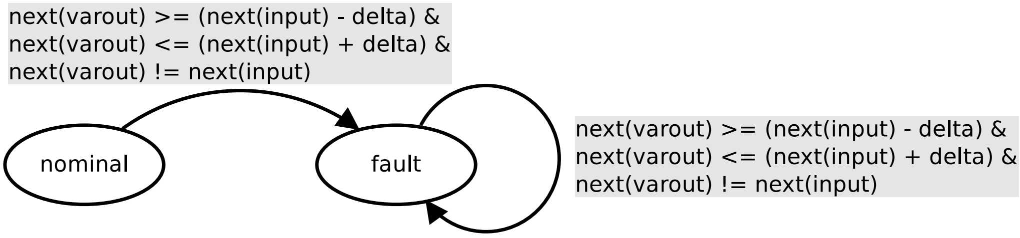


Figure 3.32: EM for a delta in erroneous by reference fault mode

Delta in erroneous by value

EM which models the effects of giving an erroneous value in a delta range from the nominal value, by value.

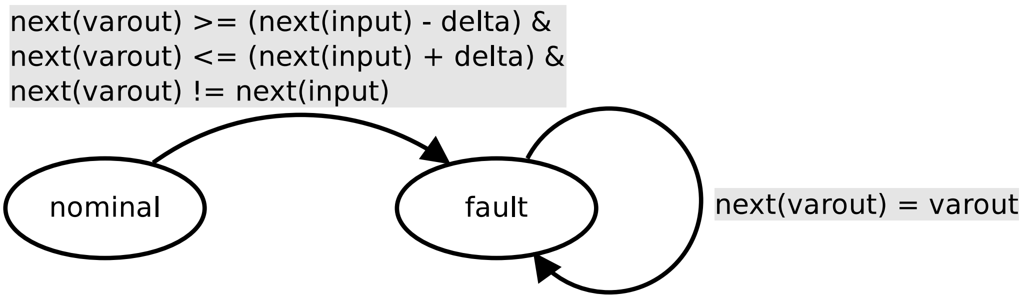


Figure 3.33: EM for a delta in erroneous by value fault mode

3.6.2 Local Dynamics

Example of a LDM which defines that a fault mode is reached when a `failure` event is non-deterministically issued, and it is transient as it can non-deterministically self-repair:

```
<local_dynamics_model name="Transient" desc="Bla bla">
  <templates>
    <template name="self_fix" type="Identifier">
      Description of the template.
    </template>
  </templates>
  <events>
    <event type="input" name="failure"/>
    <event type="input" name="${self_fix}"/>
  </events>
  <transitions>
    <transition from="nominal" to="fault">
      <trigger>failure</trigger>
    </transition>
    <transition from="fault" to="fault">
      <guard>!${self_fix}</guard>
    </transition>
    <transition from="fault" to="nominal">
      <trigger>${self_fix}</trigger>
```

```
        <guard>!failure</guard>
      </transition>
    </transitions>
</local_dynamics_model>
```

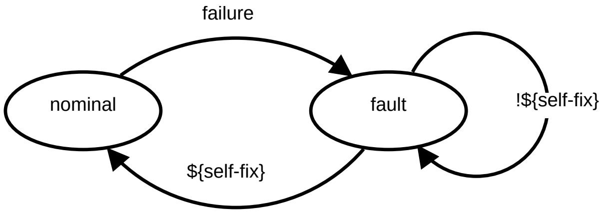


Figure 3.34: `LDM` for as self-repairing fault

Notice that the `LDM` is a template model which need to be instantiated later. `nominal` and `fault` are all reserved keywords. Like for s, keywords `is_nominal` and `is_fault` are available. Also notice that events admit negation (e.g. `!event_name`), meaning that the event is not being fired.

Chapter 4

Safety Assessment

Safety assessment allows the user to check the safety of the model, check how failure states can be reached, and which sequences of events can produce them. There are several forms of analysis, including:

1. *Fault Tree Generation* allows the user to generate a fault tree that shows the minimal sequence of events that may lead to a given undesired event (top-down analysis) – see Section 4.2.

2. *Failure Mode and Effects Analysis* links combinations of events with the list properties that may be invalidated (bottom-up analysis) – see Section 4.3.

4.1 Declaring the Fault Variables

Safety assessment can be applied to an .smv model, as long as a subset of the variables is identified as "fault variables". In case the automated fault extension is carried out, the fault variables are generated and identified automatically. On the other hand, in case the model is extended manually, the user has to specify the set of fault variables explicitly, as explained in Section 3.1. In both cases, the set of fault variables is stored as an XML file (compare Section 3.1.1).

The information on the set of fault variables is used by the underlying engines to carry out the analyses, e.g. minimal cut set computation. Fault variables are provided to the tool as an XML file. For technical reasons, such variables are *input* variables, and the tool generates a set of additional (*state*) variables that keep track of the activation of the fault variables (also called *history variables*, see [6, 5] for more details). Intuitively, a history variable becomes true as soon as the corresponding fault (variable) is triggered, and then it stays true permanently.

By default, xSAP constructs the history variables independently of the fault extension mechanism, i.e. with both the automated fault extension, and the manual fault extension. In the latter case, the user specifies an input model that is already extended and contains both the nominal and faulty behaviors.

The history variables may be already present in the extended model (e.g., in case they have been modeled manually by the user). In this scenario, they can be declared directly in the `Fault Modes` file, as explained in Section 3.1.1. There are two reasons why the user may want to declare them explicitly. First, for efficiency reasons. Second, when modeling and analyzing a static (combinational) system (i.e., a system with no transitions and only one

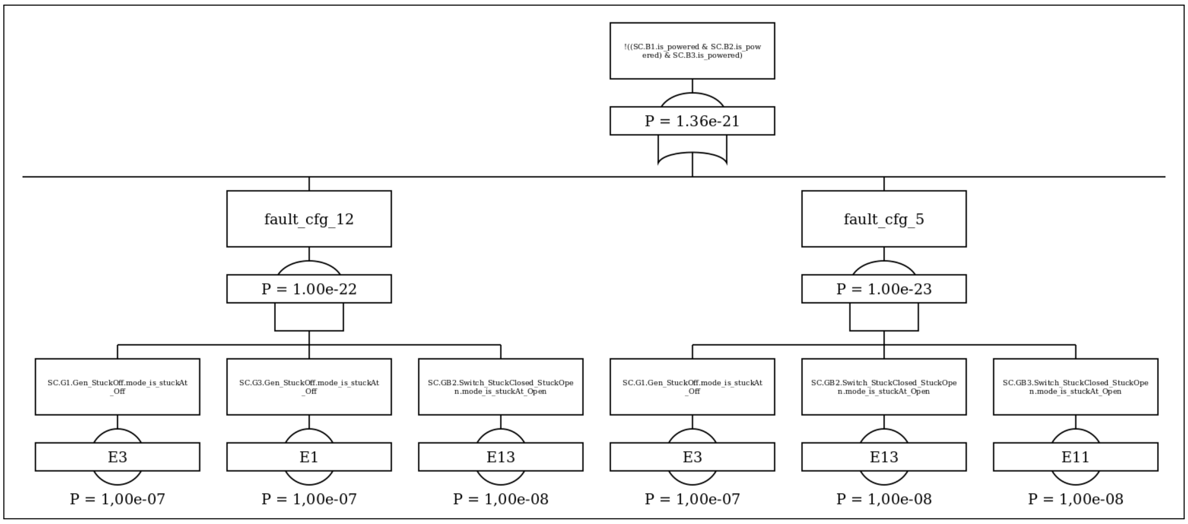


Figure 4.1: An example of fault tree

state). In this scenario, xSAP allows the user to model faults using a set of state variables that directly describe the failure states of the system, instead of modeling faults using input variables. Such set of state variables can play the role of the set of history variables.

Additional details on how to define the fault and history variables can be found in Section 3.1.1, whereas details about how to specify the fault variables, when invoking xSAP, can be found in Appendix D.1.

4.2 Fault Tree Generation

"*Fault Tree Generation*" constructs the fault tree for a given property. Basically, a fault tree is a collection of the minimal combinations of fault events that are associated with failure specified in the property.

The analysis can be done with or without hypothesis of monotonicity. Intuitively, the monotonicity hypothesis assumes that if failure is possible with a given fault configuration, then it is also possible for all the supersets. The algorithms for FT generation can be either based on BDD technology, or on SAT/SMT technology.
Remark: the algorithms based on SAT technology currently require BDDs to perform some internal manipulation of event combinations, hence, performance may be affected by the BDD variable ordering; if dynamic re-ordering of BDD variables is enabled and multiple fault trees are generated, the performance may be affected by the order in which fault trees are generated.

Once generated, the fault tree can be emitted in different output formats.

In Figure 4.1 a snippet of fault tree is shown. In this example there are:

1. two branches (called *cut sets*), each of them representing a combination of three events (linked by an 'and' gate) that together cause the given failure state

The number of events in a cut set is called *order* of the cut set.

The probabilities of the root and the internal nodes of the fault tree are computed on the basis of the probabilities of the leaves, which are specified in the `FEI` input file.

Another form of analysis is the *dynamic* fault tree generation. The difference between the dynamic fault tree generation and the fault tree generation is that the "dynamic" one shows

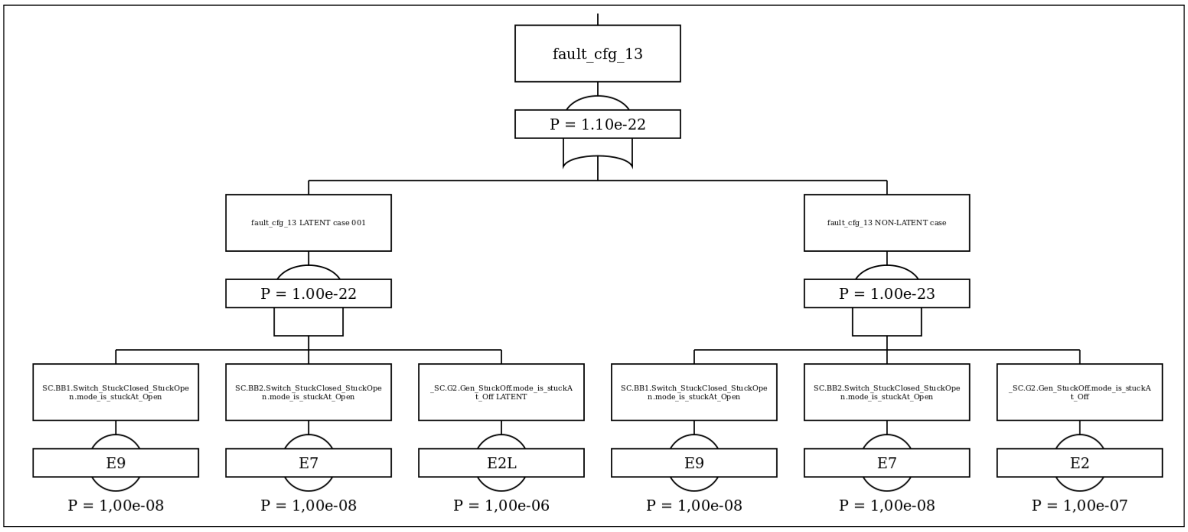


Figure 4.2: An example of fault tree with latent faults

also the precedence of the events (i.e., an event must hold before another one in order to reach the failure state) using the "priority and", in addition to the "and", gate in the fault tree.

4.2.1 Latent Faults

A fault can be declared to be possibly *latent*, that is, the corresponding item may be failed prior to the mission under analysis. Latent faults are specified in the `FEI` file, and associated with a latent probability. When latent faults are specified, they are taken into account in Fault Tree Generation, according to [25]. Namely, different cases (latent versus non-latent) are split and inserted in the fault tree. Figure 4.2 shows a snippet of a fault tree including latent faults.

4.3 Failure Modes and Effect Analysis

Failure Modes and Effect Analysis allows the user to generate, in tabular form, the set of combinations of events that may cause a given failure. If the *dynamic* option for the analysis is activated, then the order (i.e., the temporal dependence) of the events is considered important, otherwise, it is not.

The FMEA requires takes into account the properties to be checked, and the system fills a table with the results of the computation, that is then saved in various formats (e.g. comma-separated values)

As an example, an FMEA table is shown in Figure 4.3. The table contains an "*ID*" column that represents a unique identifier having the form N-M, where N identifies the fault configuration, and M the event it is associated with. The "*ID*" can be used to uniquely identify a fault configuration caused by a top level event (i.e., a single row in the FMEA table).

The highest computed cardinality can be specified in the "*cardinality*" field with command option -N. In the example this value is set to 2.

Nr	ID	Failure Mode	Failure effects
1	31-1	(_masterCC._CC1.cc & SC.G3.Gen_StuckOff.mode_is_stuckAt_Off)	!((SC.B1.is_powered & SC.B2.is_powered) & SC.B3.is_powered)
2	126-1	(_masterCC._CC1.cc & SC.GB3.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Open)	!((SC.B1.is_powered & SC.B2.is_powered) & SC.B3.is_powered)

Figure 4.3: An example of FMEA table

4.4 MTCS Analysis

Mode transition cut sets (MTCS) analysis is used to detect causes of system mode transitions, for example from an operational mode to a faulty mode with restricted functionality. The analysis is applicable on complex systems with various safety mechanisms such as redundancies and fault detection components. xSAP performs the analysis on an SMV model with a list of relevant events and a list of system modes.

The main challenge of the analysis is the formalization of the problem. Currently, xSAP implements two different approaches: strict and causal approach [7]. Both reduce the problem of finding causes to parameter synthesis problem and differ in the formula used for the parameter synthesis. The strict approach identifies events that happen in mode m_1 before transitioning to m_2. The causal approach considers also events that can happen before m_1 but excludes those that are identified as a cause of m_1. The latter approach is more sophisticated and makes use of MCS to compute causes of reaching m_1.

The analysis can be run using the provided script `compute_mtcs.py`. As input, the script takes the SMV model, an XML file with state variables represententing events in the same format as in FT analysis, and a list of expressions representing system modes.

System modes are provided in one of two possible formats: either as a list of predicate formulas over state variables in the SMV model; or as a list of state variables with Boolean or enum type. In the former, each formula represents one mode. In the latter, modes are computed as a list of all evaluations of the given variables. Based on the provided options, the script either computes MTCS for all ordered pairs of distinct modes or for transitions from the first mode to all the other provided modes. If the modes are computed from state variables, the first computed mode is considered as the source mode.

The output is given as list of sets of events for each transition. Textual output is generated in a readable XML file. Visual output can be generated either in `dot` or `tex` format (using `tikz` pictures for visualization). Examples of the outputs are given in Section 7.3.8. For the visual outputs, paging option can be used to separate transition to individual dot files or individual pages of tex document. This improves the readability in case many transitions with many cut sets are visualized.

4.5 Common Cause Analysis

Common Cause Analysis (CCA) is an important step of safety assessment. Its purpose is to evaluate the consequences of events that may break the hypothesis of independence of different faults. For instance, an engine burst may case other components of an aircraft to break simultaneously. In such case, the probability of simultaneous failure of the components

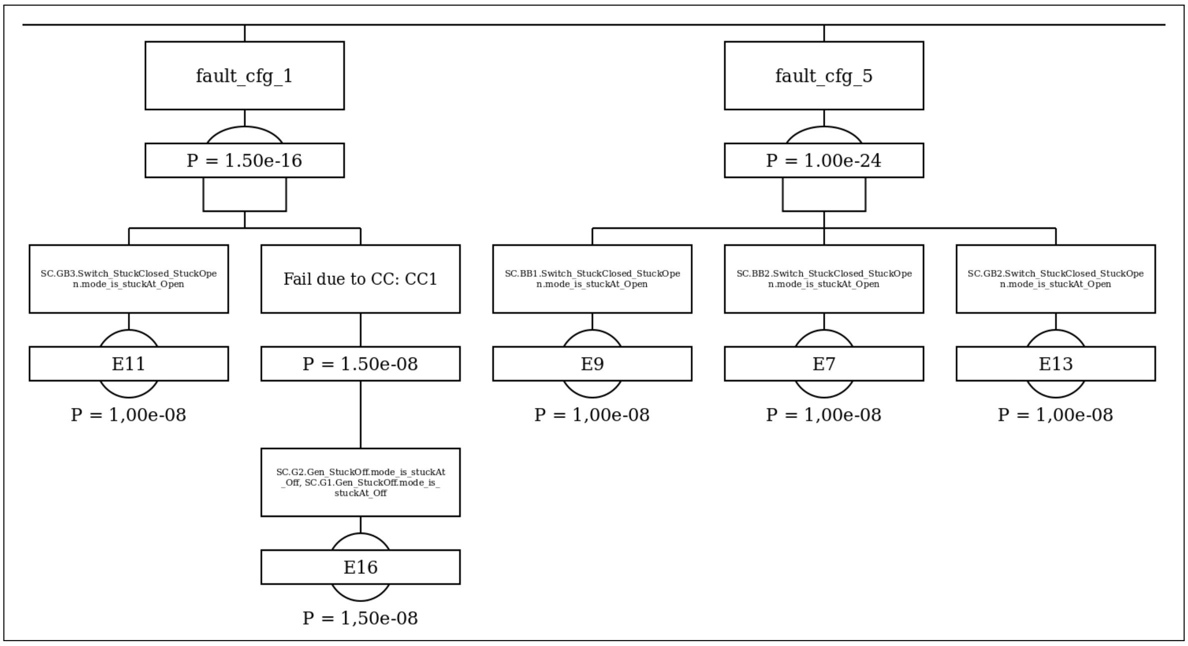


Figure 4.4: An example of fault tree with a common cause

is not given by the product of their failures as independent events – it is typically much higher.

CCA aims at investigating possible dependencies between failures, and evaluates the consequences in terms of system safety/reliability. Typically, a system design is evaluated with the goal of identifying common causes. Then, the consequences of common causes are analyzed. Finally, the impact on the design of the common causes is evaluated.

In xSAP, common cause analysis in carried out as part of Fault Tree Analysis. xSAP enables the definition of common cause events and of their consequences. Probability is attached to individual common causes, instead of (in addition to) their constituent faults as independent events. In xSAP a common cause may trigger the occurrence of a set of (dependent) faults in a user-specified manner (e.g., cascading or simultaneous).

xSAP enables the generation of FTs including common causes, and the evaluation of system reliability in presence of common causes. It is important to notice that failures due to common causes are analyzed together with component failures due to independent events (that is, independent failure of a component, and its failure due to a common cause, are both taken into account by the FT generation engine).

As an example, in Figure 4.4, the cutset on the left contains a reference to a common cause *CC1*. The `CC` is shown with the associated fault probability, and with information about the associated fault modes (*Sc.G1* and *Sc.G2* both eventually stuck at off). The cutset on the right instead shows a normal cutset containing three fault modes and no common causes.

Chapter 5

TFPG Analysis

Timed Failure Propagation Graphs (TFPG) were developed as a means to study the timed propagation of failures in complex systems [19, 18]. They represent a complementary approach to study the system's behavior under failures with respect to other safety assessment techniques such as FTA and FMECA. The focus of TFPGs is to describe the temporal dependence between a number of basic failure modes and a set of off-nominal system conditions (discrepancies) caused by the failure modes in a multi-mode (switching) system.

5.1 Timed Failure Propagation Graphs

A TFPG (Figure 5.1) consists of a labeled directed graph where nodes represent either failure modes, which are fault causes, or discrepancies, which are off-nominal conditions that are the effects of failure modes. Edges between nodes in the graph capture the effect of failure propagation over time in the underlying dynamic system. Edges in the graph model can be activated or deactivated depending on a set of possible operation modes of the system; this allows to represent failure propagation in multi-mode (switching) systems. Failure mode nodes do not have incoming edges since they are not bound to the occurrence of any other node; on the other hand, discrepancies must always be bound to the occurrence of failure modes, either directly or via intermediate discrepancies.

Time and system mode are two key factors in describing the propagation of a failure. In fact, a failure mode might lead to a chain of discrepancies that take time to occur, and a system mode might modify the way the failure propagates. For example, we might have a failure in a pipe, and this could propagate to another pipe; however, it might be the case that the two pipes are not connected (e.g., separated by a closed-valve) in the current system mode. This would mean that the failure cannot propagate from the first pipe to the second. For this reason, in a TFPG it is important to capture this information.

5.1.1 Terminology

We now fix the terminology used when talking about TFPGs as defined in [1]. Note that the original definition of TFPG did not include some features that were introduced in [1].

We say that a system may have different modes of operations, and we call them *system modes*. We assume that the system can only be in one system mode at the time and that,

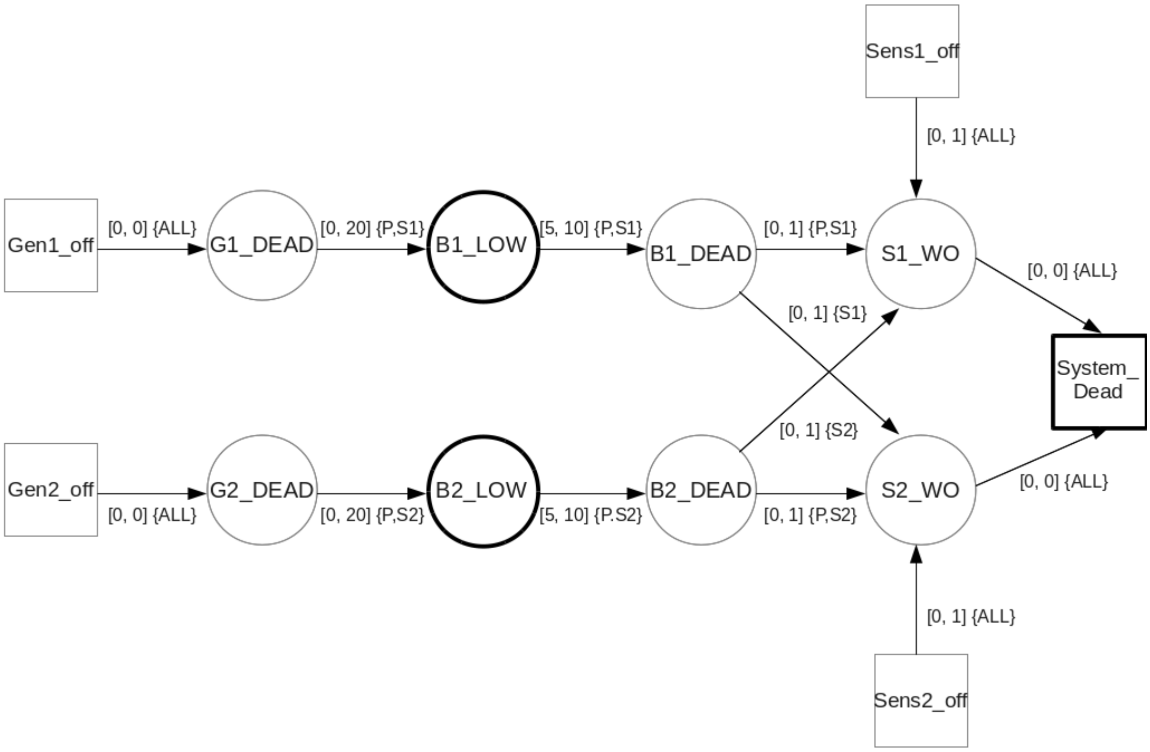


Figure 5.1: An example TFPG

given a finite amount of time, the system will change mode a finite amount of times, and that the system will stay in each mode for a non-zero amount of time.

A *failure mode* is a failure of a component of the system. A component might fail in more than one way, therefore it might have more than one failure mode. We call *fault* the occurrence of a failure of the component.

A fault in a component will produce anomalies in system behaviour. These anomalies are called *discrepancies*. We use the term *failure* to indicate a failure mode, a fault or a discrepancy.

5.1.2 TFPG Definition

We say that a TFPG is a structure $G = \langle F, D, E, M, ET, EM, DC, DS \rangle$, where:

- F is a non-empty set of failure modes;
- D is a non-empty set of discrepancies;
- $E \subseteq V \times V$ is a set of edges connecting the set of nodes $V = F \cup D$;
- M is a non-empty set of system modes. At each time instant t the system can be in only one mode;

- $ET : E \to I$ is a map that associates every edge in E with a time interval $[t_{min}, t_{max}] \in I$ indicating the min/max propagation time on the edge (where, $I \in \mathbb{R}_+ \times \mathbb{R}_+ \cup \{\inf\}$ and $t_{min} \leq t_{max}$)
- $EM : E \to \mathcal{P}(M)$ is a map that associates every edge in E with a set of modes in M. We assume that $EM(e) \neq \emptyset$ for any edge $e \in E$;
- $DC : D \to \{\texttt{AND}, \texttt{OR}\}$ is a map defining the class of each discrepancy as either `AND` or `OR` node;
- $DS : D \to \{\texttt{M}, \texttt{I}\}$ is a map defining the monitoring status of the discrepancy as either `M` for monitored or `I` not monitored (inactive).

A final assumption is that all and only failure modes can be root nodes: failure modes cannot have incoming edges, and discrepancies must have at least one incoming edge.

5.1.3 Semantics

The original semantics of TFPG [2] can be summarized as follows:

> The state of a node indicates whether the failure effects reached this node. For a failure to propagate through an edge $e = (v, v')$, the edge should be active throughout the propagation, that is, from the time the failure reaches v to the time it reaches v'. An edge e is active if and only if the current operation mode of the system mc is in the set of activation modes of the edge, that is, $mc \in EM(e)$. For an OR-type node v' and an edge $e = (v, v') \in E$, once a failure effect reaches v at time t, it will reach v' at a time t', where $e.tmin \leq t' - t \leq e.tmax$ and the edge e is active during the whole propagation. On the other hand, the activation period of an AND alarm v' is the composition of the activation periods for each link $(v, v') \in E$. When a failure propagates to a monitored node v' $(DS(v') = A)$, its physical state is considered ON; otherwise, it is OFF. If the link is deactivated any time during the propagation (because of mode switching), the propagation stops. Links are assumed memoryless with respect to failure propagation so that the current failure propagation is independent of any (incomplete) previous propagation. Also, once a failure effect reaches a node, its state will permanently change and will not be affected by any future failure propagation.

Infinity Semantics When defining the timing of a propagation, the use of ∞ can assume two different semantics: *open* and *closed*. The open semantics (written $[n, \infty)$) is used to describe the fact that the propagation time is finite, but unbounded. This means that we cannot chose a value *a-priori* to describe the maximum delay of the propagation. However, the propagation is guaranteed to occur. The closed semantics (written $[n, \infty]$) is used instead to describe propagations that might never complete (i.e., occur at infinity). This is used to capture propagation that are important, but might depend on external conditions not modeled by the TFPG (e.g., input from the environment).

xSAP only accepts TFPGs in which all infinities have an uniform semantics (being either open or close). This should be indicated in the TFPG model, for clarity. If not indicated, the default is *closed*. xSAP will inform the user if the chosen infinity semantics is not supported by the current analysis.

5.2 Reasoning Tasks

We now describe the reasoning tasks on TFPGs that are implemented in xSAP. The analyses are divided in two groups: *model-based* and *stand-alone*. The first group studies the TFPG in relation to a system model. This is used to verify that the TFPG captures the behaviors of interest, and to derive a TFPG from a system model. This group includes *Behavioral Validation*, *Synthesis*, and *Tightening*. The second group (stand-alone) aims at the study of the TFPG in isolation. It is used to prove properties on the TFPG, regardless of how the TFPG was obtained. This group includes *Possibility, Necessity, Activability and Consistency*, *Diagnosis*, *Refinement*, and *Filtering*.

The stand-alone analyses implemented in xSAP are based on [8]. These techniques are based on satisfiability modulo theory (SMT) reasoning, and provide a precise characterization of the timed behavior of the TFPG. Furthermore, they are limited to the *frozen-mode* assumption, i.e., they assume that the TFPG maintains the same mode throughout the execution.

5.2.1 Behavioral Validation

Behavioral validation is used to check if the TFPG is complete or incomplete with respect to a model [4]. This means to check whether every trace in the system model has a corresponding compatible trace in the TFPG.

This check is based on a number of “TFPG associations”, which define each failure mode, discrepancy, and system mode in terms of propositional expressions over the system state variables. If a discrepancy d is not defined in the associations file, it is defined in terms of all nodes that have an edge towards it. The semantics of an AND node that doesn’t have an associated expression is that it activates whenever all of its predecessors in the graph activate. The semantics of an OR node is that it activates whenever at least one of its predecessors has been activated. Such discrepancies are referred to as “virtual discrepancies”.

A TFPG is considered to be incomplete with respect to the system if a system trace can be found that violates the TFPG constraints. This means that there are failure propagations in the system not captured by the TFPG. Specifically, a counterexample to TFPG completeness consists of a system trace and its interpretation in terms of the TFPG variables. The system trace is a valid trace for the system model, but its TFPG interpretation does not satisfy the TFPG constraints. This can be because a discrepancy fires when it shouldn’t, for instance because none of its direct causes have occurred yet; on the other hand it can be because a discrepancy does not activate even though the TFPG constraints would require it to.

Such counterexamples can be used by the designer to get information about the missing behaviour in the TFPG, and possibly fix the TFPG or the system model, depending on where the inconsistency lies.

In addition to completeness, xSAP can also be used to test the tightness of a given TFPG w.r.t. a system model. Tightness is a property associated to the edge constraints: a TFPG is said to be tight if it is not possible for some edge to either increase its t_{min}, decrease its t_{max}, or drop any of the modes associated to the edge, without breaking completeness. In other words, a TFPG is tight if it is complete and propagations are possible in all modes and at the minimum and maximum propagation delay bounds.

We remark that the current implementation is sound but not complete for disproving the tightness of parameters $t_{max} = \infty$. Specifically, xSAP currently can identify only a subset of tightness witnesses for such assignment, and thus answer either *tight* or *unknown*.

5.2.2 Synthesis

TFPG Synthesis is used to automatically synthesize a complete TFPG, starting from a system model and a set of TFPG associations as described in Section 5.2.1. Specifically, the procedure computes the precedence constraints among all user-defined discrepancies and failure mode nodes and instantiates a respective TFPG. Where the user-defined nodes are not sufficient to express the Boolean constraints on precedences, virtual discrepancies as described in the previous section are introduced.

Note that TFPG synthesis only computes the underlying graph of the TFPG. The timings are overapproximated with tmin=0 and tmax=inf for all edges. Tight edge parameters can be identified with the tightening procedure described in Section 5.2.3.

By default the synthesis procedure analyzes, using cone-of-influence, the functional dependencies among nodes and uses this information to prune TFPG edges that represent purely temporal correlations among the events associated to nodes. This can be disabled if desired.

The following problems can appear during synthesis and will be reported to the user.

- *Unreachable Discrepancy Within Analysis Bound*: It states that a discrepancy is never activated in the model within the given bound. It is suggested to either increase the bound (if the discrepancy is expected to be activated) or remove the discrepancy from the list of associations (if it is indeed never activated). The discrepancy will still be placed in the resulting TFPG, isolated from the rest. This is for the user's convenience in debugging the problem, but results in an illegal TFPG.

- *Correlated Discrepancy Activations*: It states that there are discrepancy nodes that under certain or all failure configurations are always activated at the same time. In order to obtain a clearer graph structure it is suggested to re-run synthesis after dropping one of the correlated discrepancies from the TFPG association list given in input.

- *Spontaneous Discrepancy Activation*: It states that a case has been identified where a discrepancy node activated spontaneously, i.e. without any failure mode nodes being activated beforehand. In this case the user could redefine the discrepancy as a failure mode, investigate the expression over system state variables associated to it, or simply drop it from the problem input. The discrepancy will still be placed in the resulting TFPG, isolated from the rest. This is for the user's convenience in debugging the problem, but results in an illegal TFPG. Also note that certain TFPG simplification routines cannot be performed on such a TFPG, which can result in a graph with increased complexity.

- *Isolated Failure Modes*: When failure mode nodes appear isolated in the resulting TFPG, two cases are possible. One, the failure mode never occurs within the analysis bound. Two, the failure mode occurs but has no effect on the discrepancies within the analysis bound. The user is informed accordingly, and the failure mode node is in both cases placed into the final TFPG. The result in this case is a legal TFPG (as opposed to the case of spontaneous discrepancies), because failure mode nodes are allowed to occur spontaneously in the TFPG formalism.

5.2.3 Tightening

TFPG Tightening is used to improve the accuracy of the edge parameters in a TFPG. It takes as input a system model, association map, and a complete TFPG. The procedure tries to increase t_{min} values and decrease t_{max} values as much as possible, and drop as many mode labels as possible, without breaking the completeness of the given TFPG. A filter can be used to disable tightening of selected edge parameter classes.

Tightening of time bounds will be performed according to the highest precision of any of the time bound constants in the given TFPG. For instance, if the constant with the highest precision is 2.52, tightening considers 0.01 as the smallest possible search delta.

The t_{max} parameters that are set to $+\infty$, search for a corresponding finite tight value is limited to an upper bound. If no such bound is specified by the user, it is automatically set to the highest constant appearing in any edge constraint of the TFPG.

In cases where the user does not specify the input variable of the model encoding the time delta of each transition, only the mode labels are tightened.

5.2.4 Possibility, Necessity, Consistency, and Activability

The *Possibility* check enables the validation of a TFPG against a (partial) execution. This check can be used in different ways, in order to show additional properties of the TFPG, such as necessity, consistency and activability.

Given a TFPG and a (partial) trace of the TFPG (i.e., a set of active nodes, a set of activation times and, optionally, a mode), the *possibility check* verifies that the trace is compatible with the TFPG and provides a complete trace (i.e., a mode and an association of each node to a status and activation time). *Necessity* verifies that a partial trace is implied by the TFPG. This is usually the case when some behavior is intrinsic in the TFPG. *Consistency* is a special case of possibility, in which we check that there is at least one complete trace for the TFPG. *Activability* tests that every node can be eventually activated.

5.2.5 Refinement

When editing TFPGs, it is useful to understand the relation between the original and modified version of the TFPG. Given a (partial) mapping between the discrepancies of the original and modified TFPG, the refinement check verifies that all behaviors of the original TFPG can be mapped into behaviors of the modified one (i.e., the original TFPG *refines* the modified one).

5.2.6 Diagnosis

TFPG are commonly used to diagnose problems in systems. Given a set of timed observations, we can ask which Failure Modes could cause such a behavior. We implement two types of diagnosis. The first type simply enumerates all possible sets of failure modes (*diagnosis*) compatible with the observations. The second, instead, checks whether a given failure mode appears in all diagnosis (*certain diagnosis*). Certain diagnosis tells us whether we can be certain about the occurrence of the failure mode, given the observations. This type of diagnosis behaves particularly well with our symbolic SMT-based approach, and can be used to diagnose TFPGs with thousands of nodes.

5.2.7 Filtering

Finally, xSAP provides means to filter a given TFPG in the following two ways. First, the user might be interested in analyzing only part of a given TFPG, specifically in paths leading to selected TPFG nodes of interest. xSAP can remove all paths and nodes from the input TFPG from which the selected nodes cannot be reached. The result is a simpler more focused TFPG, on which manual inspection can be done more easily and automated analysis more efficiently. Second, when manually building a TFPG, the graph might contain redundant edges. xSAP provides the means to automatically remove such edges if present.

5.3 TFPG Formats

TFPG files and TFPG association files use two different types of format:

- Textual Format
- XML Format

5.3.1 Textual Format

We first describe the concrete syntax, i.e. the textual format.

TFPG file

The textual format for TFPG files (extension: `.tfpg`) makes use of the following keywords:

- "*NAME*": the name of TFPG
- "*FM*": the Failure Mode (FM node_name)
- "*AND*": the AND Discrepancy (AND node_name)
- "*OR*": the OR Discrepancy (OR node_name)
- "*EDGE*": the Edge (EDGE edg_name src_node dst_node min_time max_time edge_modes)
- "*MONITORED*": indicates if a discrepancy is monitored
- "*INFINITY*": indicates infinite time
- "*ALL*": all the modes
- "*MODES*": the Modes (mode_name)
- "*INFINITY_SEMANTICS_OPEN*", "*INFINITY_SEMANTICS_CLOSED*": indicates which semantics to use for ∞

We remark that:

- Comments are those lines starting with '`--`'

- Identifiers (name of the TFPG, name of a node, name of a mode and name of an edge) are sequences of the following characters: `'A'..'Z'`, `'a'..'z'`, `'0'..'9'`, `'_'`, `'-'`

From a high-level point of view, a structural TFPG consists of a list of nodes, which can be of three different types:

- **Failure Modes**
- **Discrepancies:**
  - **And discrepancies**
  - **Or discrepancies**

and of a list of modes and a list of edges connecting those nodes.
Each Node is associated to:

- **Type:** the type of the node (Failure Mode, And Discrepancy, Or Discrepancy)
- **Name:** the name of the node
- **IsMonitored:** a value stating whether it is monitored or not

Each Edge is associated to:

- **SrcNode:** the name of the source node
- **DestNode:** the name of the destination node
- **Name:** the name of the edge
- **TMin:** the minimum time of the edge
- **TMax:** the maximum time of the edge
- **Modes:** a list of modes

An example of the textual file is given in Figure 5.2.
A textual file can be written in free format. It must contain:

1. "*Name*" of the TFPG: it is the first (non-empty) token of the file, which consists of the keyword "*NAME*" followed by the name of the TFPG
2. (Optionally) The semantics used for the "*INFINITY*" operator (OPEN or CLOSED). If not specified, the default is Closed.
3. Definition of the "*nodes*". A node is defined through a keyword defining its type "*(FM, AND or OR)*" followed by its name and a value stating whether it is monitored or not. For instance, the following are possible definitions:

```
FM Gen1_off
AND System_Dead MONITORED
OR G1_DEAD
```

```
NAME Sensor-Generator

INFINITY_SEMANTICS_CLOSED

FM Gen1_off
FM Gen2_off
FM Sens1_off
FM Sens2_off
AND System_Dead MONITORED
OR G1_DEAD
OR B1_LOW MONITORED
OR B1_DEAD
OR S1_WO
OR G2_DEAD
OR B2_LOW MONITORED
OR B2_DEAD
OR S2_WO

MODES Primary,Secondary1,Secondary2

EDGE EDGE1 Gen1_off G1_DEAD 0 INFINITY (Primary,Secondary1,Secondary2)
EDGE EDGE2 G1_DEAD B1_LOW 0 INFINITY (Primary,Secondary1)
EDGE EDGE3 B1_LOW B1_DEAD 0 INFINITY (Primary,Secondary1)
EDGE EDGE4 B1_DEAD S1_WO 0 INFINITY (Primary,Secondary1)
EDGE EDGE5 S1_WO System_Dead 0 INFINITY (Primary,Secondary1,Secondary2)
EDGE EDGE6 Gen2_off G2_DEAD 0 INFINITY (Primary,Secondary1,Secondary2)
EDGE EDGE7 G2_DEAD B2_LOW 0 INFINITY (Primary,Secondary2)
EDGE EDGE8 B2_LOW B2_DEAD 0 INFINITY (Primary,Secondary2)
EDGE EDGE9 B2_DEAD S2_WO 0 INFINITY (Primary,Secondary2)
EDGE EDGE10 S2_WO System_Dead 0 INFINITY (Primary,Secondary1,Secondary2)
EDGE EDGE11 B1_DEAD S2_WO 0 INFINITY (Secondary1)
EDGE EDGE12 B2_DEAD S1_WO 0 INFINITY (Secondary2)
EDGE EDGE13 Sens1_off S1_WO 0 INFINITY (Primary,Secondary1,Secondary2)
EDGE EDGE14 Sens2_off S2_WO 0 INFINITY (Primary,Secondary1,Secondary2)
```

Figure 5.2: TFPG textual format example

4. Definition of the "*modes*". This is a part related to the edge definition. Modes are introduced by the keyword "*MODES*" and are comma-separated, like for instance:

```
MODES Primary, Secondary1, Secondary2
```

5. Definition of the edges. An edge is defined through a keyword ("*EDGE*"), followed by its name, the source node, the destination node, the minimum time, the maximum time and the list of the modes, defined as a comma-separated list encapsulated in parentheses. Also in this case, two choices as the ones presented for nodes are possible. For instance, the following are possible definitions:

```
EDGE EDGE1 Gen1_off G1_DEAD 0 INFINITY ALL
EDGE EDGE2 G1_DEAD B1_LOW 0 INFINITY (Primary, Secondary1)
EDGE EDGE3 B1_LOW B1_DEAD 0 INFINITY (Primary, Secondary1)
```

We remark that:

- Using the keyword "*INFINITY*" the maximum time can be defined to be infinity, e.g:

```
EDGE EDGE15 Gen1_off G1_DEAD 0 INFINITY (Primary)
```

- In the current implementation TMin values other than 0 are ignored and internally replaced by 0, and TMax values other than infinity are ignored and internally replaced by 0. Support of timing will be implemented in a future release.
- An edge can be active in all possible modes; in this case, instead of explicitly writing all the existent modes, the user can use the keyword "*ALL*", e.g:

```
EDGE EDGE13 Sens1_off S1_WO 0 INFINITY ALL
```

TFPG Associations file

The textual format for TFPG associations (extension: `.tfpga`) contains the following keywords:

- "FAILURE_MODES": to introduce the list of expressions related to failure modes.
- "MONITORED_DISCREPANCIES": to introduce the list of expressions related to monitored discrepancies.
- "UNMONITORED_DISCREPANCIES": to introduce the list of expressions related to unmonitored discrepancies.
- "MODES": to introduce the list of expressions related to the TFPG modes.

Each keyword is followed by the list of associations of that type; two assumptions are done in this case:

1. The identifier (i.e. the TFPG symbol) is the first word encountered in a line consisting of alphanumeric, underscore ('_') or minus ('-') characters.
2. The related expression is given by every character found starting from the identifier enclosed between the space character (' ') and newline ('backslash n').

An example of this format is given in Figure 5.3.

5.3.2 XML Format

A sample TFPG file (whose name, by convention, ends with `.txml`) looks as follows:

```
<?xml version='1.0' encoding='UTF-8'?>
<tfpg name="Sensor-Generator">
  <nodesList>
    <node name="Gen1_off" isMonitored="false">
      <type>FM</type>
```

```
FAILURE_MODES
 Gen1_off root.sc_sys.sc_psu1.sc_generator.sc__errorSubcomponent.mode = error_dead
 Gen2_off root.sc_sys.sc_psu2.sc_generator.sc__errorSubcomponent.mode = error_dead
 Sens1_off root.sc_sys.sc_sensor1.sc__errorSubcomponent.mode = error_dead
 Sens2_off root.sc_sys.sc_sensor2.sc__errorSubcomponent.mode = error_dead

MONITORED_DISCREPANCIES
B1_LOW root.sc_sys.sc_psu1.sc_battery.data_low
B2_LOW root.sc_sys.sc_psu2.sc_battery.data_low
System_Dead !root.sc_sys.data_is_alive

UNMONITORED_DISCREPANCIES
G2_DEAD root.sc_sys.sc_psu2.sc_generator.data_has_power = FALSE
G1_DEAD root.sc_sys.sc_psu1.sc_generator.data_has_power = FALSE
S1_WO root.sc_sys.sc_sensor1.data_reading = FALSE
S2_WO root.sc_sys.sc_sensor2.data_reading = FALSE
B1_DEAD root.sc_sys.sc_psu1.sc_battery.data_has_power_out = FALSE
B2_DEAD root.sc_sys.sc_psu2.sc_battery.data_has_power_out = FALSE

MODES
Primary root.sc_sys.data_mode_selector = const_Primary
Secondary1 root.sc_sys.data_mode_selector = const_Secondary1
Secondary2 root.sc_sys.data_mode_selector = const_Secondary2
```

Figure 5.3: TFPG Associations textual format example


```
</node>
<node name="Gen2_off" isMonitored="false">
  <type>FM</type>
</node>
<node name="Sens1_off" isMonitored="false">
  <type>FM</type>
</node>
<node name="Sens2_off" isMonitored="false">
  <type>FM</type>
</node>
<node name="System_Dead" isMonitored="true">
  <type>AND</type>
</node>
<node name="G1_DEAD" isMonitored="false">
  <type>OR</type>
</node>
<node name="B1_LOW" isMonitored="true">
  <type>OR</type>
</node>
<node name="B1_DEAD" isMonitored="false">
  <type>OR</type>
</node>
<node name="S1_WO" isMonitored="false">
```

```
    &lt;type&gt;OR&lt;/type&gt;
  &lt;/node&gt;
&lt;/nodesList&gt;
&lt;modesList&gt;
  &lt;mode&gt;Primary&lt;/mode&gt;
  &lt;mode&gt;Secondary1&lt;/mode&gt;
  &lt;mode&gt;Secondary2&lt;/mode&gt;
&lt;/modesList&gt;
&lt;edgesList&gt;
  &lt;edge name="EDGE1"&gt;
    &lt;srcNode&gt;Gen1_off&lt;/srcNode&gt;
    &lt;tMin&gt;0.0&lt;/tMin&gt;
    &lt;tMax&gt;-1.0&lt;/tMax&gt;
    &lt;modesList&gt;
      &lt;mode&gt;Primary&lt;/mode&gt;
      &lt;mode&gt;Secondary1&lt;/mode&gt;
      &lt;mode&gt;Secondary2&lt;/mode&gt;
    &lt;/modesList&gt;
    &lt;destNode&gt;G1_DEAD&lt;/destNode&gt;
  &lt;/edge&gt;
  &lt;edge name="EDGE2"&gt;
    &lt;srcNode&gt;G1_DEAD&lt;/srcNode&gt;
    &lt;tMin&gt;0.0&lt;/tMin&gt;
    &lt;tMax&gt;-1.0&lt;/tMax&gt;
    &lt;modesList&gt;
      &lt;mode&gt;Primary&lt;/mode&gt;
      &lt;mode&gt;Secondary1&lt;/mode&gt;
    &lt;/modesList&gt;
    &lt;destNode&gt;B1_LOW&lt;/destNode&gt;
  &lt;/edge&gt;
  &lt;edge name="EDGE3"&gt;
    &lt;srcNode&gt;B1_LOW&lt;/srcNode&gt;
    &lt;tMin&gt;0.0&lt;/tMin&gt;
    &lt;tMax&gt;-1.0&lt;/tMax&gt;
    &lt;modesList&gt;
      &lt;mode&gt;Primary&lt;/mode&gt;
      &lt;mode&gt;Secondary1&lt;/mode&gt;
    &lt;/modesList&gt;
    &lt;destNode&gt;B1_DEAD&lt;/destNode&gt;
  &lt;/edge&gt;
  &lt;edge name="EDGE4"&gt;
    &lt;srcNode&gt;B1_DEAD&lt;/srcNode&gt;
    &lt;tMin&gt;0.0&lt;/tMin&gt;
    &lt;tMax&gt;-1.0&lt;/tMax&gt;
    &lt;modesList&gt;
      &lt;mode&gt;Primary&lt;/mode&gt;
```

```
        <mode>Secondary1</mode>
      </modesList>
      <destNode>S1_WO</destNode>
    </edge>
    <edge name="EDGE5">
      <srcNode>S1_WO</srcNode>
      <tMin>0.0</tMin>
      <tMax>-1.0</tMax>
      <modesList>
        <mode>Primary</mode>
        <mode>Secondary1</mode>
        <mode>Secondary2</mode>
      </modesList>
      <destNode>System_Dead</destNode>
    </edge>
  </edgesList>
</tfpg>
```

The value “-1” is used to symbolize infinity for TMax values.

A sample TFPG associations file (whose name, by convention, ends with `.axml`) looks as follows:

```
<?xml version='1.0' encoding='UTF-8'?>
<associations>
  <failureModes>
    <assoc id="Sens1_off"
           expr="root.sc_sys.sc_sensor1.sc__errorSubcomponent.mode = error_dead"/>
    <assoc id="Sens2_off"
           expr="root.sc_sys.sc_sensor2.sc__errorSubcomponent.mode = error_dead"/>
  </failureModes>
  <monitoredDiscrepancies>
    <assoc id="B1_LOW" expr="root.sc_sys.sc_psu1.sc_battery.data_low"/>
    <assoc id="B2_LOW" expr="root.sc_sys.sc_psu2.sc_battery.data_low"/>
    <assoc id="System_Dead" expr="!root.sc_sys.data_is_alive"/>
  </monitoredDiscrepancies>
  <unmonitoredDiscrepancies>
    <assoc id="S1_WO" expr="root.sc_sys.sc_sensor1.data_reading = FALSE"/>
    <assoc id="S2_WO" expr="root.sc_sys.sc_sensor2.data_reading = FALSE"/>
  </unmonitoredDiscrepancies>
  <tfpgModes>
    <assoc id="Primary" expr="root.sc_sys.data_mode_selector = const_Primary"/>
    <assoc id="Secondary1" expr="root.sc_sys.data_mode_selector = const_Secondary1"/>
    <assoc id="Secondary2" expr="root.sc_sys.data_mode_selector = const_Secondary2"/>
  </tfpgModes>
</associations>
```

Chapter 6

Fault Detection And Isolation

Fault Detection and Isolation (*FDI*) is concerned with diagnosing the faulty behavior of a system, identifying the specific fault that has occurred and designing a system able to activate suitable alarms whenever given faults occur.

This analysis can be organized in four different activities:

Diagnosability analysis : the model of the system is analyzed to identify whether a specific fault (or combinations thereof) is diagnosable, i.e., there exists an ideal diagnoser that is always able to detect the fault.

Generation of minimum observables set : this is an optional step in which the minimum set of observables required for detecting a specified fault is synthesized.

Synthesis of diagnoser : using automatic synthesis techniques, a diagnoser is generated that raises an alarm every time a given fault occurs.

Effectiveness analysis : the diagnoser is validated by model checking properties on the combined model (i.e., the system model + the model of the diagnoser). It is possible to use the diagnoser automatically generated with synthesis techniques, or manually design it.

6.1 Diagnosability Analysis

The purpose of diagnosability analysis is to verify whether enough observables are available to always detect a specific condition within a given time bound.

In order to perform this analysis we first need to specify the *diagnosis condition* of interest - for instance the occurrence of a fault - together with a desired on the *diagnosis delay bound*. Furthermore we need to specify a file specifying the *observable* variables needs to be given. These variables are the signals we can monitor in the system and which will be available to the diagnoser. Optionally, a *diagnosis context* can be specified to exclude certain traces that are unrealistic. This is relevant when the model doesn't include, for instance, precise description of the environment or of a controller for the modeled system. Note that if the condition is diagnosable under a context there is no guarantee w.r.t. behaviors outside the context. In practice this means that a diagnoser synthesized for the specification is guaranteed to raise the alarm only on traces covered by the context.

A counterexample to diagnosability, called *critical pair*, consists of a pair of traces that are observationally indistinguishable. On one trace the condition occurs, and on the other trace it doesn't (within the required time bound). Since the observations are the same on both traces, it is impossible for a diagnoser to distinguish them and thus raise the alarm with confidence that the fault actually happened.

A full description of the framework can be found in [9].

6.2 Generation of minimum observables set

The purpose of minimum observables set generation is to identify subsets of observables that still guarantee diagnosability of the chosen condition. This is helpful when trying to identify what sensors capture sufficient information, and in practice also to simplify the FDI implementation and possibly reduce its cost. This analysis is carried out by using the same inputs presented in diagnosability analysis section, i.e. a diagnosis condition, a delay bound, a set of available observables, and optionally a diagnosis context. Note that generation of minimum observables set is an optional step, which is not strictly required in order to complete FDI analysis.

6.3 Synthesis of diagnoser

The purpose of synthesis is to generate a diagnoser able to raise an alarm every time a specific fault occurs.

This analysis is carried out given a model of the system to be analyzed, the set of observables, and the specification of the alarms to be raised. In particular, if a fault is diagnosable in a given context, the synthesized diagnoser will be able to raise the alarm whenever the fault has occurred and the condition on the context (i.e., the indicator event) is satisfied.

The synthesis routines are based on the notions of:

- *Trace diagnosability*: the diagnosis is localized to individual traces (executions). The diagnoser will raise an alarm if it knows for sure that the fault has occurred in a specific execution, otherwise it will not.
- *Maximality of the diagnoser*: the diagnoser will raise an alarm *as soon as* it knows for sure that the fault has occurred.

Notice that we can specify many different alarms; a possible approach consists in adding an alarm for each different possible fault, and another one for a generic fault, represented by the disjunction of all possible faults.

6.4 Effectiveness analysis

Effectiveness analysis consists in the validation of the generated diagnoser.

The generated diagnoser is encoded in a module containing a state variable, representing the different states of it; additionally, three *DEFINE* constraints are contained, whose meaning is as follows:

1. *KFAULT*: represents the disjunction of those states where the diagnoser knows that the fault named FAULT has occurred.

2. *KnFAULT*: represents the disjunction of those states where the diagnoser knows that the fault named FAULT has not occurred.

3. *UFAULT*: represents the disjunction of those states where the diagnoser cannot decide whether the fault named FAULT has occurred or not.

By model checking a set of automatically generated LTL properties containing the previous *DEFINE* constraints, we can validate the generated diagnoser.

6.5 Files format

6.5.1 Observables file

The observables file simply lists (one per line) the variables representing the observable part of the system.

A sample observables file looks as follows:

```
SC.G1.state
cmd_G1
SC.G2.state
cmd_G2
```

Figure 6.1: Observables file example

6.5.2 Alarm Specification file

The alarm specification file lists the set of alarms to be considered. The suggested extension for these files is *.asl*.

Each alarm is defined with the following fields:

NAME The name of the alarm (mandatory)

CONDITION The diagnosis condition associated with the alarm (mandatory)

TYPE The type of the alarm: finite, bounded, exact (optional – defaults to finite)

DELAY The delay associated with the alarm. This makes sense only if type is bounded or exact.

CONTEXT The LTL context to consider when studying the alarm (optional)

```
NAME: A2
CONDITION: noise=1

NAME: A2
CONDITION: mb.ox
TYPE: exact
DELAY: 1

NAME: A5
CONDITION: mb.ox
TYPE: bounded
DELAY: 5
CONTEXT: G F TRUE
```

Figure 6.2: ASL file example

Chapter 7

Triple Generator Example

In this document we use as a running example a model of a triple redundant generator. Its characteristics are described in the following sections.

7.1 Informal Description

7.1.1 The Plant

The triple generator example, represented in figure 7.1, is composed of several components which are described below:

- Generators (G1, G2 and G3): they have a single output link and can have two possible states:
  - "*ON*": the generator provides energy on the output link;
  - "*OFF*": the generator is switched off;
- Circuit Breakers (GB1, GB2, GB3, BB1, BB2 and BB3): they have two electric links and can have two possible states:
  - "*OPEN*": the circuit breaker maintains separated the two electric links;
  - "*CLOSED*": the circuit breaker connects the two electric links to each other;
- Buses (B1, B2 and B3): they represent the loads that can be powered by the generators. Each Bus has three links always connected each other so they must have the same value at each step;
- Controller: the component is in closed-loop with the system, which is sensed/controlled by the Controller itself.

System Faults

Both the generators and the circuit breakers change their internal state ("*ON*" or "*OFF*" for generators and "*OPEN*" or "*CLOSED*" for circuit breakers) according to the signal provided by the controller. This behavior is called nominal but it can diverge in case of failure as follows:

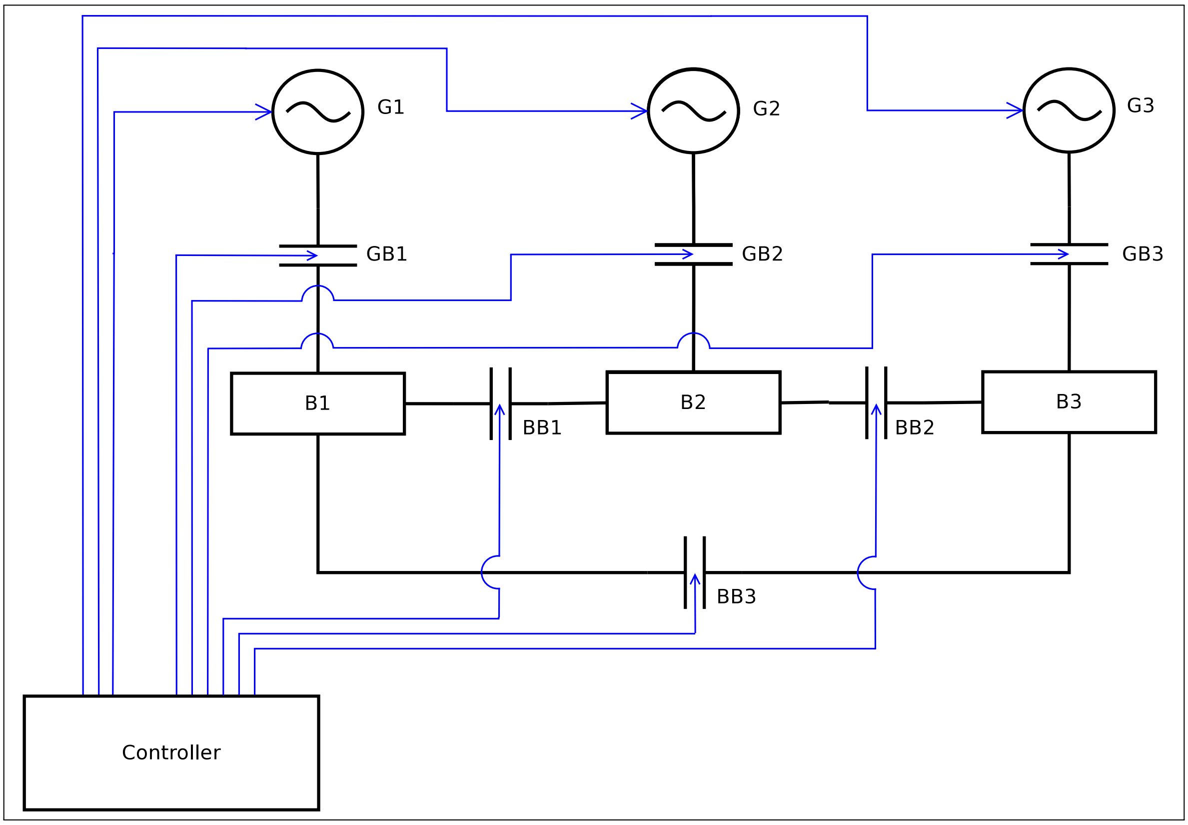


Figure 7.1: Triple Generator Example

- Generators can fail off permanently. It means that when the fault occurs the internal state of the generator is stuck at "*OFF*" permanently;
- Circuit breakers can fail open and closed. It means that, depending on the specific fault, the internal state of the circuit breaker is stuck at "*OPEN*" or "*CLOSED*" but the fault is not permanent, so the component can go back to the nominal behavior.

7.1.2 Controller behavior

Source to bus priority

The triple generator system has redundancies paths from the generators to each bus, this means that it is possible to provide the power in different way. In order to have a policy that privileges some power configuration respect to others they are defined some priority constraints on the Generator-to-Bus power, shown in table 7.1, and also the priority for each Generator-to-Bus path as depicted in table 7.2.

7.1.3 System Requirements

The behavior of the controller has to guarantee the observance of the following requirements:

1. No bus will be connected to more than 1 power source at any time.
2. If any power source is on, then all buses will be powered.

BUS	High priority	Medium priority	Low priority
B1	G1	G2	G3
B2	G2	G1	G3
B3	G2	G1	G3

Table 7.1: Bus power source priority

Source to bus paths	Priority	B1	B2	B3
G1	High	NA	BB1	BB3
	Low	NA	BB3-BB2	BB1-BB2
G2	High	BB1	NA	BB2
	Low	BB2-BB3	NA	BB1-BB3
G3	High	BB3	BB2	NA
	Low	BB2-BB1	BB3-BB1	NA

Table 7.2: Source to bus path priority

3. Bus power source priority and source to bus path priority schemes will be respected at all time (see tables 7.1 and 7.2).

4. If no power source is on, then all buses will be unpowered.

5. Any single/dual component failure will not cause other system requirements to be violated.

6. Never more than two generators on, unless required in case of failures.

7.2 SMV modeling

The modeling of the Triple Generator system is composed of two main components: the controller and the system configuration.

The system configuration (see code 7.4) links together generators (code 7.1), switches (code 7.2) and buses (code 7.3) and it describes how they interact.

```
MODULE Generator(cmd, init_state)
  VAR
    state : {on, off};

  DEFINE
    is_on := (state = on);

  DEFINE
    is_off := (state = off);

  ASSIGN
    next(state) :=
      case
```

```
        (cmd = cmd_on) : on;
        (cmd = cmd_off) : off;
        TRUE : state;
      esac;

  ASSIGN
   init(state) := init_state;
```

Code 7.1: Module Generator

```
MODULE Switch(cmd, init_state)
  VAR
    state : {open, closed};

  DEFINE
    is_closed := (state = closed);
    is_open := (state = open);

  ASSIGN
   next(state) :=
     case
       (cmd = cmd_closed) : closed;
       (cmd = cmd_open) : open;
       TRUE : state;
     esac;

  ASSIGN
    init(state) := init_state;
```

Code 7.2: Module Switch

```
MODULE Bus(in1, in2, in3)
  VAR
    state : {working, broken};

  DEFINE
    is_broken := (state = broken);
  DEFINE
    is_powered := (state = working) & (count((in1), (in2), (in3)) = 1);

  -- When the bus is overpowered it becomes broken and unfixable --
  ASSIGN
    init(state) :=
      case
        (count((in1), (in2), (in3)) > 1) : broken;
        TRUE : working;
      esac;

    next(state) :=
      case
        next((count((in1), (in2), (in3)) > 1)) : broken;
        TRUE : state;
      esac;
```

Code 7.3: Module Bus

```
MODULE System_Configuration(cmd_Gs, cmd_CBs)
   DEFINE
    init_G1 := on;
    init_G2 := on;
    init_G3 := off;

    init_GB1 := closed;
    init_GB2 := closed;
    init_GB3 := open;
    init_BB1 := open;
    init_BB2 := closed;
    init_BB3 := open;

  VAR
    G1 : Generator(cmd_Gs[index_G1], init_G1);
    G2 : Generator(cmd_Gs[index_G2], init_G2);
    G3 : Generator(cmd_Gs[index_G3], init_G3);

    GB1 : Switch(cmd_CBs[index_GB1], init_GB1);
    GB2 : Switch(cmd_CBs[index_GB2], init_GB2);
    GB3 : Switch(cmd_CBs[index_GB3], init_GB3);

    BB1 : Switch(cmd_CBs[index_BB1], init_BB1);
    BB2 : Switch(cmd_CBs[index_BB2], init_BB2);
    BB3 : Switch(cmd_CBs[index_BB3], init_BB3);

  -- Bus instances, see below for the input definition --

    B1 : Bus(B1_poweredby_G1, B1_poweredby_G2, B1_poweredby_G3);
    B2 : Bus(B2_poweredby_G1, B2_poweredby_G2, B2_poweredby_G3);
    B3 : Bus(B3_poweredby_G1, B3_poweredby_G2, B3_poweredby_G3);

  -- Definition of the signals (in matrix format) used by Controller
  -- to know the state of the single components

  DEFINE
    init_Gs := [
      init_G1,
      init_G2,
      init_G3];

    init_CBs := [
      init_GB1,
      init_GB2,
      init_GB3,
      init_BB1,
      init_BB2,
      init_BB3];

  DEFINE
    index_GB1 := 0;
    index_GB2 := 1;
    index_GB3 := 2;

    index_BB1 := 3;
    index_BB2 := 4;
    index_BB3 := 5;

    index_G1 := 0;
    index_G2 := 1;
    index_G3 := 2;

  DEFINE
    event_Gs := [
      fev_off_Gs,
      nev_Gs];
```

```
  event_CBs := [
    fev_open_CBs,
    fev_closed_CBs,
    nev_CBs];

DEFINE
  fev_open_CBs := [
    GB1.fev_stuck_at_open,
    GB2.fev_stuck_at_open,
    GB3.fev_stuck_at_open,
    BB1.fev_stuck_at_open,
    BB2.fev_stuck_at_open,
    BB3.fev_stuck_at_open];

  fev_closed_CBs := [
    GB1.fev_stuck_at_closed,
    GB2.fev_stuck_at_closed,
    GB3.fev_stuck_at_closed,
    BB1.fev_stuck_at_closed,
    BB2.fev_stuck_at_closed,
    BB3.fev_stuck_at_closed];

  nev_CBs := [
    GB1.nev,
    GB2.nev,
    GB3.nev,
    BB1.nev,
    BB2.nev,
    BB3.nev];

DEFINE
  fev_off_Gs := [
    G1.fev_stuck_at_off,
    G2.fev_stuck_at_off,
    G3.fev_stuck_at_off];

  nev_Gs := [
    G1.nev,
    G2.nev,
    G3.nev];


-- Definition of the possible paths to Bus 1 --

DEFINE
  B1_poweredby_G1_U := (G1.is_on) & (GB1.is_closed);
  B1_poweredby_G2_L := B3_poweredby_G2_L & (BB3.is_closed);
  B1_poweredby_G2_R := B2_poweredby_G2_U & (BB1.is_closed);
  B1_poweredby_G3_L := B3_poweredby_G3_U & (BB3.is_closed);
  B1_poweredby_G3_R := B2_poweredby_G3_R & (BB1.is_closed);

-- Definition of the possible paths to Bus 2 --

DEFINE
  B2_poweredby_G1_L := B1_poweredby_G1_U & (BB1.is_closed);
  B2_poweredby_G1_R := B3_poweredby_G1_R & (BB2.is_closed);
  B2_poweredby_G2_U  := (G2.is_on) & (GB2.is_closed);
  B2_poweredby_G3_L := B1_poweredby_G3_L  & (BB1.is_closed);
  B2_poweredby_G3_R := B3_poweredby_G3_U & (BB2.is_closed);

-- Definition of the possible paths to Bus 3 --

DEFINE
  B3_poweredby_G1_L := B2_poweredby_G1_L & (BB2.is_closed);
  B3_poweredby_G1_R := B1_poweredby_G1_U & (BB3.is_closed);
  B3_poweredby_G2_L := B2_poweredby_G2_U & (BB2.is_closed);
```

```
    B3_poweredby_G2_R := B1_poweredby_G2_R & (BB3.is_closed);
    B3_poweredby_G3_U := (G3.is_on) & (GB3.is_closed);


-- Definition of the possible inputs for Bus 1 --

DEFINE
    B1_poweredby_G1 := B1_poweredby_G1_U;
    B1_poweredby_G2 := B1_poweredby_G2_R | B1_poweredby_G2_L;
    B1_poweredby_G3 := B1_poweredby_G3_R | B1_poweredby_G3_L;

-- Definition of the possible inputs for Bus 2 --

DEFINE
    B2_poweredby_G1 := B2_poweredby_G1_R | B2_poweredby_G1_L;
    B2_poweredby_G2 := B2_poweredby_G2_U;
    B2_poweredby_G3 := B2_poweredby_G3_R | B2_poweredby_G3_L;

-- Definition of the possible inputs for Bus 3 --

DEFINE
    B3_poweredby_G1 := B3_poweredby_G1_R | B3_poweredby_G1_L;
    B3_poweredby_G2 := B3_poweredby_G2_R | B3_poweredby_G2_L;
    B3_poweredby_G3 := B3_poweredby_G3_U;
```

Code 7.4: Module Circuit_System

A complete analysis of the Triple Redundant Generator example is out of the scope of this document. The full example is available in the xSAP tool distribution. In this manual, only some parts are reported, in order to illustrate the modeling in SMV language.

7.3 Concrete example of Fault Extension

7.3.1 Nominal Model

This example is taken from a larger model, but in this context only two modules are interesting: module `Generator` and module `Switch`.

Here a selected extract is presented, for seeing the complete example see:
file `examples/fe/triple_modular_generator/SC_TMG.smv`
and associated `FEI` file `examples/fe/triple_modular_generator/SC_TMG.fei`

Module `Generator` can break and can propagate a failure through event `fev_stuck_at_off`.

```
MODULE Generator(cmd, init_state)
  VAR state : {on, off};

  -- Definition of the transition labels between nominal and fault state
  IVAR
    fev_stuck_at_off : boolean;
    nev : boolean;

  TRANS nev = FALSE;

  DEFINE
    is_on := (state = on);

  DEFINE
    is_off := (state = off);

```

```
ASSIGN
  next(state) :=
    case
      (cmd = cmd_on) : on;
      (cmd = cmd_off) : off;
      TRUE : state;
    esac;

ASSIGN
 init(state) := init_state;
```

Module `Switch` can break and can propagate a failure through events `fev_stuck_at_closed` and `fev_stuck_at_open`.

```
MODULE Switch(cmd, init_state)
VAR
  state : {open, closed};

-- Definition of the transition labels among nominal and fault states
IVAR
  fev_stuck_at_closed : boolean;
  fev_stuck_at_open : boolean;
  nev : boolean;

DEFINE is_closed := (state = closed);
DEFINE is_open := (state = open);

ASSIGN
  next(state) :=
    case
      (cmd = cmd_closed) : closed;
      (cmd = cmd_open) : open;
      TRUE : state;
    esac;

ASSIGN
  init(state) := init_state;
```

7.3.2 Fault Extension Instruction

We want modules `Generator` and `Switch` to be affected by faults, for the moment with no common causes involved.
In particular, `Generator` shall be affected by a stuck-at effect (to `off` value), permanently. For this fault one `StuckAtByValue_D fm` with `Permanent LDM` does the job:

```
FAULT EXTENSION FE_SC_TMG
  /-- ... --/

  EXTENSION OF MODULE Generator

     /-- Description of Fault Slice Gen_StuckOff --/
     SLICE Gen_StuckOff AFFECTS state WITH

       /-- Description of fault mode stuckAt_Off --/
       MODE stuckAt_Off : Permanent StuckAtByValue_D(
                   data term << off,
                   data input << state,
                   data varout >> state,
                   event failure >> fev_stuck_at_off);
```

Here data `value` is assigned to constant `off`, and `Generator`'s variable `state` is the `AS`, and within the `fm` it is read through `input` and written through `varout`.
`LDM`'s events `failure` is bound to `Generator`'s event `fev_stuck_at_off`, and template `self_fix` is bound to `Generator`'s event `nev`.

Notice that parameters are bounded to the corresponding affected symbols by using the read operator `<<` and the write operator `>>`:

```
data input << state
data varout >> state
```

In this example, value of variable `state` in the `NC` is bounded to read value `input` and write value `varout`. This means that current value of `state` can be read through `input`, and can be written through `varout`.
Templates are instantiated by using the `=` operator.

Module `Switch` shall be affected either by a stuck-at-closed or by a stuck-at-open faults. This requires two `fm`s both transient. Furthermore, a `GDM` is defined to make possible move from `stuck-open` to `stuck-closed` by issuing event `failure` in `fm stuckAt_Closed`.

```
FAULT EXTENSION FE_SC_TMG
  /-- ... --/

  EXTENSION OF MODULE Switch

     /-- Description of Fault Model for Switch --/
     SLICE Switch_StuckClosed_StuckOpen
      AFFECTS state WITH

        /-- Description of fault mode StuckAt_Closed --/
        MODE stuckAt_Closed : Transient StuckAtByValue_D(
                   data term << closed,
                   data input << state,
                   data varout >> state,
                   template self_fix = self_fixed,
                   event failure >> fev_stuck_at_closed,
                   event self_fixed >> nev);

        /-- Description of fault mode StuckAt_Open --/
```

```
      MODE stuckAt_Open : Transient StuckAtByValue_D(data term << open,
                 data input << state,
                 data varout >> state,
                 template self_fix = self_fixed,
                 event failure >> fev_stuck_at_open,
                 event self_fixed >> nev);

      GLOBAL DYNAMICS
        /-- Transition of FM Global Dynamics --/
        TRANS stuckAt_Open.fault -[stuckAt_Closed.failure]-> stuckAt_Closed.fault;
```

Code 7.5: `FEI` for the Switch module

Here two `fm`s are defined, and notice the particular way transitions in the `GDM` are defined. There are two special keywords `nominal` and `fault` which identify the corresponding locations in each `fm`.

7.3.3 Modules and Module Instances in FEI

In the previous `FEI` example, *all* instances of modules `Generator` and `Switch` were affected. If we want to affect a subset of their instances (or different instances in different ways), construct `FOR INSTANCES` has to be used:

```
FAULT EXTENSION FE_SC_TMG

  EXTENSION OF MODULE Generator

     /-- affects only SC.G1 and SC.G3 --/
     FOR INSTANCES SC.G[13]

     /-- ... --/


  EXTENSION OF MODULE Switch

     SLICE Switch_StuckClosed_StuckOpen

        /-- affects SC.GB1, and all SC.BBs (SC.BB1, SC.BB2, SC.BB3) --/
        FOR INSTANCES SC.GB1, SC.BB*
        AFFECTS state WITH
        /-- ... --/
```

When specifying instances, list can be provided, and each entry may contain wild-card characters (*?[...]).

Instances can be defined at two different levels:

Module (above, done for `Generator`) Affects specified instances for all slices which do not specify instances explicitly.

Slice (above, done for `Switch`) Affects specified instances for a single slice, *overriding* any instance specification done for the containing module (if any).

7.3.4 Properties

With reference to the example proposed in Chapter 7, the requirements violation can be checked by defining some properties. We illustrate some examples below.

R2: if any power source is on, then all buses will be powered This requirement can be translated into an INVARSPEC as follows:

```
(G1.is_on | G2.is_on | G3.is_on) ->
(B1.is_powered & B2.is_powered & B3.is_powered)
```

R4: if no power source is on, then all buses will be unpowered This requirement can be translated into an INVARSPEC as follows:

```
(G1.is_off & G2.is_off & G3.is_off) ->
(!B1.is_powered & !B2.is_powered & !B3.is_powered)
```

7.3.5 Formal properties

As described in the NUXMV user manual [20], the property definition is supported at modeling level in the following format:

```
[INVAR,LTL,CTL]SPEC NAME <property_name> := <property>;
```

7.3.6 Choose Fault Templates

With reference to the example proposed in Chapter 7, there is the following faults situation:

- Generators can fail off permanently;
- Circuit Breakers can fail open or closed transitorily.

This can be obtained by modeling two Fault Slices, one for each affected variable. With reference to the faults library defined in the appendix 3.6, the parameters needed by the faults library used to cover the faulty behavior of Generators and Circuit Breakers are represented in tables 7.3 and 7.4. The Fault Slice of Generators has a single Fault Mode, as depicted in table 7.3, and this implies that it is not necessary to have a Global Dynamics description.

Differently from the Slice Model of Generators, the Circuit Breakers need the definition of two Fault Modes, one for each faulty behavior (fail open and fail closed) as depicted in figures 7.2(a) and 7.2(b). The transient dynamics for Circuit Breakers imposes to have an event that occurs when the system goes back to nominal case. In the case proposed in figure 7.2 that event is called "*self_fix_event*" and it can be used at controller level if the controller has a full observability of the system state and configuration, and this is the case proposed in the Triple Generator Example (see Chapter 7).

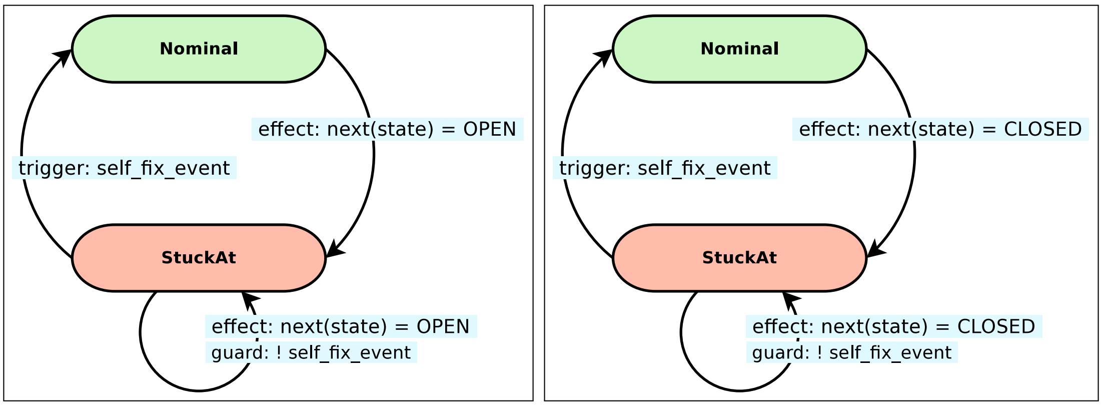
Module	**Affected Variable**	**Effect Model**		**Local Dynamics**
Generator	state	StuckAtByValue_D	OFF	Permanent

Table 7.3: Fault Modes of the Generator

With reference to the example proposed in Chapter 7, in Figure 7.1, `FEI` code in 7.5 shows the Fault Slice of the Circuit Breaker.

Module	Affected Variable	Effect Model		Local Dynamics
Circuit Breaker	state	StuckAtByValue_D	OPEN	Transient
Circuit Breaker	state	StuckAtByValue_D	CLOSED	Transient

Table 7.4: Fault Modes of the Circuit Breakers



(a) StuckAt_Open (b) StuckAt_Closed

Figure 7.2: Fault Slice of Circuit Breaker

7.3.7 Result of Fault Extension

To create the extended model, run the Model Extender script:

```
$> cd examples/fe/triple_modular_generator/
$> python ../../../scripts/extend_model.py -v \
        SC_TMG.smv SC_TMG.fei

INFO:root:Generated XML fei: out/SC_TMG.xml
INFO:root:Generating expanded XML fei: out/expanded_SC_TMG.xml
INFO:root:Generated expanded XML fei: out/expanded_SC_TMG.xml
INFO:root:Generated xsap commands script: out/xsap_extend_model.cmd
INFO:root:Invoking xsap to carry out smv extension of SC_TMG.smv
INFO:root:Successfully created:
INFO:root: - Extend smv file: 'out/extended_SC_TMG.smv'
INFO:root: - Fault modes xml file: 'out/fms_SC_TMG.xml'
```

Script `extend_model.py` has option -d which allows to specify the output directory to put all generated files into (default: `./out`).

As shown by the verbose messages (enabled by option -v), generated files are:

- file `out/extended_SC_TMG.smv`
- file `out/fms_SC_TMG.xml`

Both will be used to perform the Safety Assessment Analysis in the following section.

7.3.8 Safety Assessment

Generating Fault Trees

In this example, the good property we want to study is "All of the buses B1, B2 and B3 are powered.", which corresponds to a TLE "At least one bus B1, B2 or B3 is not powered."

To generate a fault tree, some options are available. Here is an instance:

```
$> pwd
.../examples/fe/triple_modular_generator

$> python ../../../scripts/compute_ft.py -v \
     --smv-file out/extended_SC_TMG.smv \
     --fms-file out/fms_SC_TMG.xml \
     --prop-text '(!SC.B1.is_powered | \
                   !SC.B2.is_powered | \
                   !SC.B2.is_powered)' \
     --engine ic3 -b
INFO:root:Invoking xsap to compute fault tree
INFO:root:xsap produced fault tree in:
INFO:root:  events: 'out/extended_SC_TMGevents.txt'
INFO:root:  gates: 'out/extended_SC_TMGgates.txt'
```

Remark: notice the use of single straight quotes to protect special characters for bash shell (e.g. character !). Other shells may require different quotation characters.

The generated fault tree can then be shown with the Fault Tree Viewer:

```
$> python ../../../scripts/view_ft.py -v \
     --events-file out/extended_SC_TMGevents.txt \
     --gates-file out/extended_SC_TMGgates.txt \
```

The Fault Tree Viewer opens and shows the fault tree:

Figure 7.3 shows one cut set where:

- G1 is stuck-at-off
- GB2 is stuck-at-open
- GB3 is stuck-at-open

In this scenario, none of the buses B1, B2 and B3 is powered.

Many cut sets are found (one is depicted in figure 7.3). Details can be found in the output provided by xSAP which is run by the script.

A total of 26 fault trees are found, and the minimal cardinality is 3 (14 minimal cut sets with cardinality 3, and 12 with cardinality 4):

```
$> cat out/xsap_compute_ft.out

...
***************  STATISTICS  *************
```

Order	Number	Cumul
0	0	0
1	0	0
2	0	0
3	14	14
4	12	26

Command `compute_ft.py` offers many options, and in particular it can be used with different engines. It can also show (-s) the generated fault tree by invoking the Fault Tree Viewer automatically when done.

Execute `compute_ft.py -h` to see all available options.

Generating FMEA Tables

Generating a FMEA table takes one or more TLEs as inputs, and produces one file in CSV format, with tab as separator among fields.

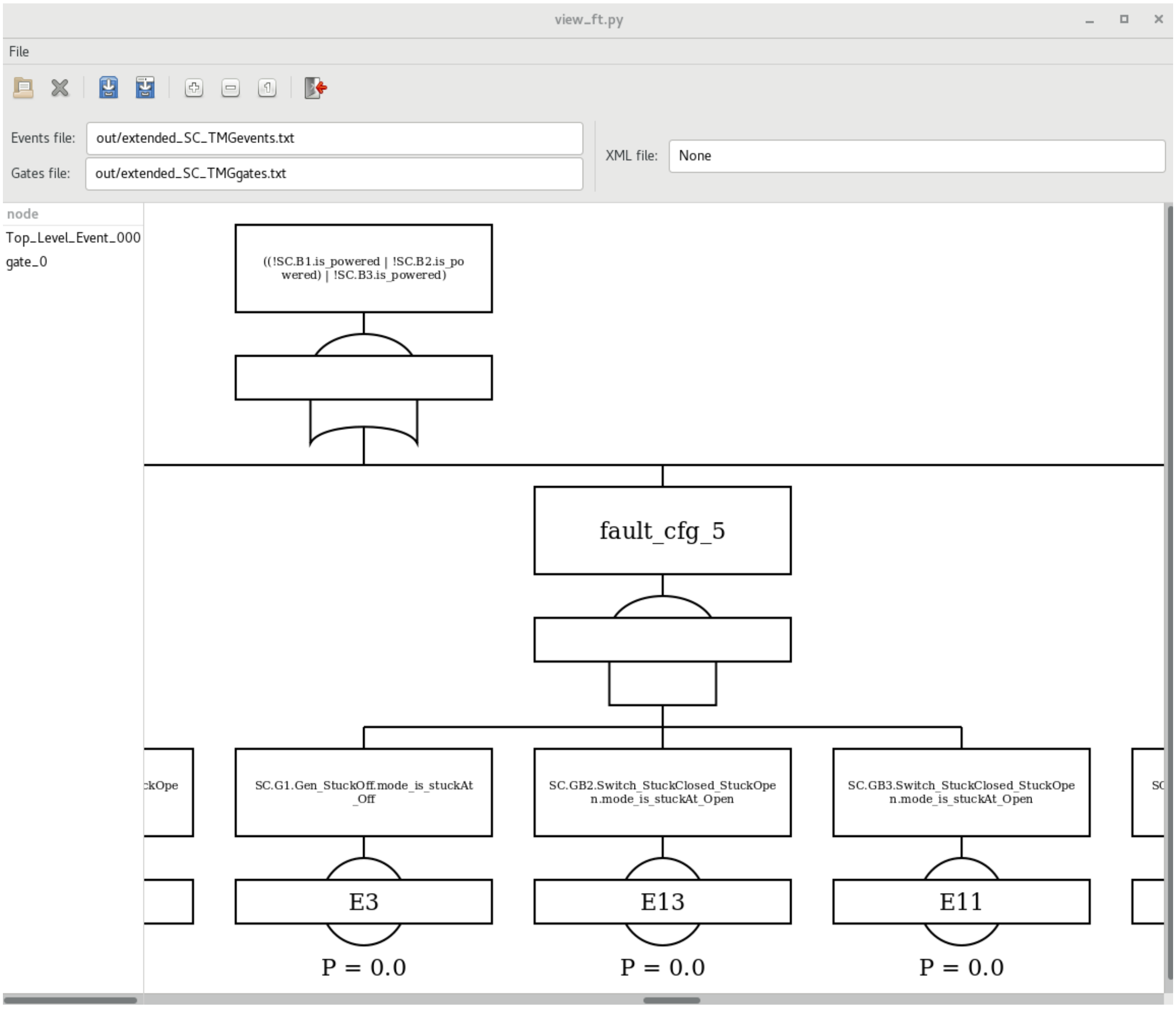


Figure 7.3: Fault Tree for TLE: “At least a bus is not powered”

There are four fields:

1. Incremental number of the cut set
2. ID of the cut set
3. Failure Modes
4. TLE

In our example, the property set is limited to the same TLE we used for generating the fault tree: "At least one bus `B1`, `B2` or `B3` is not powered."

We assume that the extended model has been already built into directory `./out` as done in previous steps.

From the Fault Tree, we already know that the minimal fault cardinality is 3, however we begin with a cardinality 1 (option -N):

```
$> pwd
.../examples/fe/triple_modular_generator

$>

$> python ../../../scripts/compute_fmea_table.py \
      --props-text='(!SC.B1.is_powered   |
                     !SC.B2.is_powered   |
                     !SC.B3.is_powered)' -N 1 -v

INFO:root:Invoking xsap to compute fmea table
INFO:root:xsap produced fmea table in: 'out/extended_SC_TMGfmea_table.txt'
WARNING:root:The FMEA table is empty with cardinality 1
```

The command generates an empty FMEA table as expected. Now we will increase the cardinality to 3, and ask to show the table when done (-s):

```
$> python ../../../scripts/compute_fmea_table.py \
      --props-text='(!SC.B1.is_powered   |
                     !SC.B2.is_powered   |
                     !SC.B3.is_powered)' -N 3 -v -s

INFO:root:Invoking xsap to compute fmea table
INFO:root:xsap produced fmea table in: 'out/extended_SC_TMGfmea_table.txt'
--------------------------------------------------------------------------------
ID:  1
TLE: ((!SC.B1.is_powered | !SC.B2.is_powered) | !SC.B3.is_powered)
FMS: (SC.G1.Gen_StuckOff.mode_is_stuckAt_Off = TRUE & \
     (SC.G2.Gen_StuckOff.mode_is_stuckAt_Off = TRUE & \
      SC.G3.Gen_StuckOff.mode_is_stuckAt_Off = TRUE))
--------------------------------------------------------------------------------
ID:  2
TLE: ((!SC.B1.is_powered | !SC.B2.is_powered) | !SC.B3.is_powered)
FMS: (SC.GB1.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Open = TRUE & \
     (SC.G2.Gen_StuckOff.mode_is_stuckAt_Off = TRUE & \
      SC.G3.Gen_StuckOff.mode_is_stuckAt_Off = TRUE))
--------------------------------------------------------------------------------
...
--------------------------------------------------------------------------------
ID:  14
TLE: ((!SC.B1.is_powered | !SC.B2.is_powered) | !SC.B3.is_powered)
FMS: (SC.GB1.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Open = TRUE & \
     (SC.GB2.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Open = TRUE & \
      SC.GB3.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Open = TRUE))
================================================================================
Total cut sets with cardinality 3: 14
```

Notice that the number of cut sets is equal to the number of cut sets found in previously computed fault tree with cardinality 3. This happens since a FMEA table with cardinality N includes the table with cardinality $N-1$, and since the FMEA tables with cardinalities 1 and 2 are empty.

This is the result of increasing the cardinality to 4:

```
$> python ../../../scripts/compute_fmea_table.py \
      --props-text='(!SC.B1.is_powered   |
                     !SC.B2.is_powered   |
                     !SC.B3.is_powered)' -N 4 -v -s
```

```
INFO:root:Generated xsap commands script: out/xsap_compute_fmea.cmd
INFO:root:Invoking xsap to compute fmea table
INFO:root:xsap produced fmea table in: 'out/extended_SC_TMGfmea_table.txt'
-----------------------------------------------------------------------
ID:  1
TLE: ((!SC.B1.is_powered | !SC.B2.is_powered) | !SC.B3.is_powered)
FMS: (SC.G1.Gen_StuckOff.mode_is_stuckAt_Off = TRUE & \
     (SC.G2.Gen_StuckOff.mode_is_stuckAt_Off = TRUE & \
      SC.G3.Gen_StuckOff.mode_is_stuckAt_Off = TRUE))
-----------------------------------------------------------------------
ID:  2
TLE: ((!SC.B1.is_powered | !SC.B2.is_powered) | !SC.B3.is_powered)
FMS: (SC.GB1.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Open = TRUE & \
     (SC.G2.Gen_StuckOff.mode_is_stuckAt_Off = TRUE & \
      SC.G3.Gen_StuckOff.mode_is_stuckAt_Off = TRUE))
-----------------------------------------------------------------------
...
-----------------------------------------------------------------------
ID:  179
TLE: ((!SC.B1.is_powered | !SC.B2.is_powered) | !SC.B3.is_powered)
FMS: (SC.GB1.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Open = TRUE & \
     (SC.GB1.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Closed = TRUE & \
     (SC.GB2.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Open = TRUE & \
      SC.GB3.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Open = TRUE)))
=======================================================================
Total cut sets with cardinality 4: 179
```

Total number of cut sets increased to 179, while for fault tree it was 26 $(14 + 12)$. This is because FMEA tables are not minimal like cut sets in fault trees, and in particular they consider all possible faults even when (some) may be not really causing the TLE directly. For example with cardinality 4 you will find cut sets containing 3 faults causing the problem, and a fourth fault not really contributing, like a stuck-at-closed fault for a switch:

```
ID:  91
TLE: ((!SC.B1.is_powered | !SC.B2.is_powered) | !SC.B3.is_powered)
FMS: (SC.GB1.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Closed = TRUE & \
     (SC.GB3.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Open = TRUE & \
     (SC.G1.Gen_StuckOff.mode_is_stuckAt_Off = TRUE & \
      SC.G2.Gen_StuckOff.mode_is_stuckAt_Off = TRUE)))
```

E.g. in the found cut set the fault:

```
SC.GB1.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Closed
```

does not contribute to the TLE as generator `G1` is broken off anyway.

As for `compute_ft.py`, command `compute_fmea_table.py` offers many options, and in particular it can be used with different engines. Execute `compute_fmea_table.py -h` to see all available options.

Generating MTCS

We show the computed MTCS for some selected mode transitions in the running example of the triple generator model. As system modes we consider all configurations of the generator modes. That is, for each G_1, G_2, G_3 we get state either `on` or `off`. There are 8 system modes in total, hence 56 transitions of distinct system modes. The SMV model is extended with a new state variable `mode`, whose value corresponds to the actual combination of generator modes.

The xSAP command `compute_mode_transition_cut_sets` is called by a provided python script `compute_mtcs.py`. Examples for running this python script are in Listings 7.4, 7.5, 7.6. In Listing 7.4, all 56 transitions are analyzed and output is written in `output.xml` file in a readable form, and in `output.tex` file, where each transition is visualized on a separate page. The `tex` file needs to be built by LuaLaTeX to generate the `output.pdf` file with 56 pages. In Listing 7.5, only one transition is analyzed. Again, XML and tex output are generated. The visualized MTCS for this transition is shown in Figure 7.7(a). In Listing 7.6, two transitions are analyzed. The textual XML output is shown in Listing 7.8 and dot visualization in Figure 7.7(b).

```
$> python compute_mtcs.py --smv-file extended_SC_TMG.smv
                          --fms-file fms_SC_TMG.xml
                          -V -g -o tex --expressions mode
```

Figure 7.4: Example of a command to compute MTCS for all mode transitions. Option `-V` specifies that expression is a variable.

```
$> python compute_mtcs.py --smv-file extended_SC_TMG.smv
                          --fms-file fms_SC_TMG.xml
                          -g -e -o tex
                          --expressions "mode␣=␣on_on_off" "mode␣=␣on_off_off"
```

Figure 7.5: Example of a command to compute MTCS for the mode transition from `on_on_off` to `on_off_off`. `-e` specifies that only transitions from the first mode to all the other are considered.

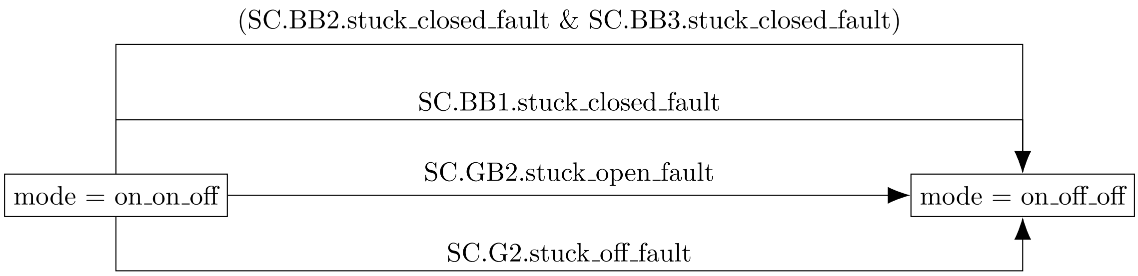
```
$> python compute_mtcs.py --smv-file extended_SC_TMG.smv
                          --fms-file fms_SC_TMG.xml
                          -e -o dot
                          --expressions "mode␣=␣on_off_off"
                                        "mode␣=␣off_on_off"
                                        "mode␣=␣off_off_on"
```

Figure 7.6: Example of a command to compute MTCS for the mode transition from `on_on_off` to `on_off_off` and the mode transition from `on_on_off` to `off_off_on`.

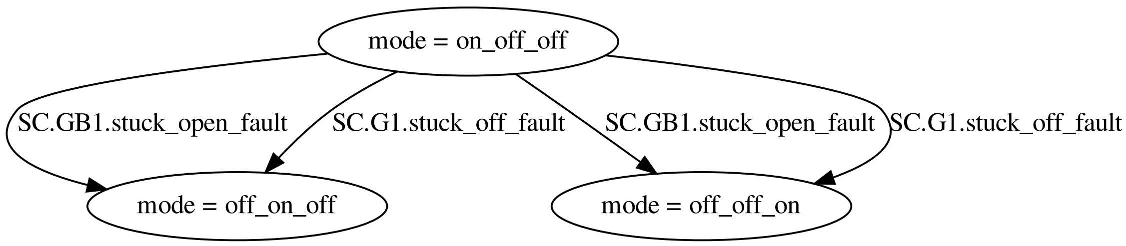
7.3.9 Adding Common Cause

We now add `CC` behaviour, by adding three `CC` to our `FEI` specification:

```
COMMON CAUSES

  /-- ---------------------------------------------------------- --/
  /-- Failure of G1 followed by failure of G3, which lead to a
      switch failure --/
  CAUSE CC1
        MODULE Generator
          FOR INSTANCES SC.G1
          MODE Gen_StuckOff.stuckAt_Off WITHIN 0 .. 0;

        MODULE Switch
          FOR INSTANCES SC.GB2
          MODE Switch_StuckClosed_StuckOpen.stuckAt_Open WITHIN 3 .. 5;

        MODULE Generator
          FOR INSTANCES SC.G3
          MODE Gen_StuckOff.stuckAt_Off WITHIN 1 .. 3;

  /-- ---------------------------------------------------- --/
  /-- Instantaneous failure of generators G1 and G2        --/
  CAUSE CC2
        MODULE Generator
          FOR INSTANCES SC.G[12]
          MODE Gen_StuckOff.stuckAt_Off WITHIN 0 .. 0;

  /-- ---------------------------------------------------- --/
  /-- Simultaneous failure of all generators               --/
```



(a) Visual `tex` output of MTCS found by command in Listing 7.5.



(b) Visual `dot` output of MTCS found by command in Listing 7.6.

Figure 7.7: Example of visual outputs of MTCS.

```
    CAUSE CC3
        MODULE Generator
          MODE Gen_StuckOff.stuckAt_Off WITHIN 1 .. 2;
```

Cause `CC1` refers two Generators and one Switch, supposing that a failure break off of instance `G1`, may lead to a failure of `G3` as well in 1 to 3 steps, which may lead to a failure of switch `GB2` in the following 3 steps (3..5).

Cause `CC2` refers two Generators (`G1 and G2`), meaning that both may be involved in a single failure called `CC2`.

Cause `CC3` models a simultaneous failure of all instances of module `Generator`, as the `MODULE` part of the `CC` specification does not specify the instances explicitly, *all* instances which have been extended are intended to be affected together.

When dealing with the TLE used in the example ("At least one bus `B1`, `B2` or `B3` is not powered"), notice that:

- `CC1` produces a cut set with cardinality 1, as it breaks all power lines from generators.
- `CC2` produces two cut set with cardinality 2, as a stand-alone failure to the line coming from `G3` is needed together with `CC2`.
- `CC3` produces a cut set with cardinality 1, as all generators are involved.

By adding only `CC2` to the specification, this is the generated FMEA table:

```
<mtcs>
    <transition>
        <from_mode>mode = on_off_off</from_mode>
        <to_mode>mode = off_on_off</to_mode>
        <cutsets>
            <cutset>
                <event>SC.GB1.stuck_open_fault</event>
            </cutset>
            <cutset>
                <event>SC.G1.stuck_off_fault</event>
            </cutset>
        </cutsets>
    </transition>
    <transition>
        <from_mode>mode = on_off_off</from_mode>
        <to_mode>mode = off_off_on</to_mode>
        <cutsets>
            <cutset>
                <event>SC.GB1.stuck_open_fault</event>
            </cutset>
            <cutset>
                <event>SC.G1.stuck_off_fault</event>
            </cutset>
        </cutsets>
    </transition>
</mtcs>
```

Figure 7.8: XML output of MTCS found by command in Listing 7.6.

```
$> python ../../../scripts/extend_model.py -v -d out_cc \
      SC_TMG.smv SC_TMG_CC.fei

INFO:root:Generated XML fei: out_cc/SC_TMG_CC.xml
INFO:root:Generating expanded XML fei: out_cc/expanded_SC_TMG_CC.xml
INFO:root:Generated expanded XML fei: out_cc/expanded_SC_TMG_CC.xml
INFO:root:Generated xsap commands script: out_cc/xsap_extend_model.cmd
INFO:root:Invoking xsap to carry out smv extension of SC_TMG.smv
INFO:root:Successfully created:
INFO:root: - Extend smv file: 'out_cc/extended_SC_TMG.smv'
INFO:root: - Fault modes xml file: 'out_cc/fms_SC_TMG.xml'

$> python ../../../scripts/compute_fmea_table.py -d out_cc \
     --props-text='(!SC.B1.is_powered  |
                    !SC.B2.is_powered  |
                    !SC.B3.is_powered)' -N 2 -v -s
INFO:root:Invoking xsap to compute fmea table
INFO:root:xsap produced fmea table in: 'out_cc/extended_SC_TMGfmea_table.txt'
--------------------------------------------------------------------------------
ID:  1
TLE: ((!SC.B1.is_powered | !SC.B2.is_powered) | !SC.B3.is_powered)
FMS: (_masterCC._CC2.cc = TRUE & SC.G3.Gen_StuckOff.mode_is_stuckAt_Off = TRUE)
--------------------------------------------------------------------------------
ID:  2
TLE: ((!SC.B1.is_powered | !SC.B2.is_powered) | !SC.B3.is_powered)
FMS: (_masterCC._CC2.cc = TRUE & SC.GB3.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Open = TRUE)
================================================================================
Total cut sets with cardinality 2: 2
```

The result is as expected involving two failures, `CC2` and a failure of `G3` or `GB3` in order to break power supply coming from `G3`.

7.3.10 Adding Fault Probability

A numerical probability can be associated to each `fm` and `CC`. From the grammar:

```
<fault-mode> ::=
    MODE <mode-id> (probability-value)? ':'
         <local-dynamics-model-id> <effect> ';'

<common-cause> ::=
    CAUSE <id> (probability-value)?
    (<cc-module-modes>)+

<probability-value> ::=
     '{' <real-number> '}'
  |  '{' prob:<real-number> '}'
```

`real-number` can be a number ($0 \leq N \leq 1$) like e.g.:

- `0.123`
- `123.e-4`, or `123.E-4` (*IMPORTANT: notice the use of '.'*)
- `123.e-4`, or `123.4e-5`, or `0.001e+2`, or `0.001e2`

Precision is limited to 15 digits at the moment.

The value of probability can be given as a raw number, or it can be given as a named parameter called `prob`. The raw form is supported for backward compatibility, while the latter form is preferrable and should be used instead.

For example, `prob:0.123e-3`.

To associate a probability value to e.g. a Fault Mode of each extended instances of `Generator`:

```
      MODE stuckAt_Off {1.e-7}: Permanent StuckAtByValue_D(...);
```

To associate a probability value to e.g. Common Cause `CC1`:

```
    CAUSE CC1 {1.5e-8}
        MODULE Generator
         ...
```

When no probability is specified, `0` is assumed.

Remark: specifying that a `fm` has `0` probability does not disable the `fm` itself; the probability computed for the cut sets containing that `fm` will be set to be `0`, but those cut sets will still appear in fault trees and FMEA tables.

When extending the model, the produced Fault modes xml file will contain probability information associated to each single module instance which is affected by the extension. E.g:

```
<?xml version="1.0"?>
<compass>
  <fmlist>
    <fm name="_SC.G3.Gen_StuckOff.mode_is_stuckAt_Off" nominal_value="FALSE" probability="1.e-7"/>
    <fm name="_SC.G2.Gen_StuckOff.mode_is_stuckAt_Off" nominal_value="FALSE" probability="1.e-7"/>
    <fm name="_SC.G1.Gen_StuckOff.mode_is_stuckAt_Off" nominal_value="FALSE" probability="1.e-7"/>
    <fm name="_masterCC._CC1.cc" nominal_value="FALSE" probability="1.5e-8"/>
  </fmlist>

  <obslist>
  </obslist>
</compass>
```

The analysis will use this XML file as input, so changing probability values (on a per-instance basis) within it will propagate values to the analysis results.

7.3.11 Latent Faults

Being *latent* is a property of fault modes. A Fault Mode can be declared to be *possibly latent* (with an associated latent probability) with:

```
<fault-mode> ::=
    MODE <mode-id> (probability-value-mode)? <colon>
         <local-dynamics-model-id> <effect> <semi-colon>

<probability-value-mode> ::=
    <lbra> <real-number> (, latent:yes|no, latent_prob:<real-number>)?<rbra>
    | <lbra> prob:<real-number> (, latent:yes|no, latent_prob:<real-number>)?<rbra>
```

The latent-property is specified along with the Fault Mode probability. For example:

```

EXTENSION OF MODULE Generator
   SLICE Gen_StuckOff AFFECTS state WITH

      MODE stuckAt_Off {prob:1.e-7, latent:yes, latent_prob:1.e-6} :
         Permanent StuckAtByValue_D(...);
```

In this example `stuckAt_Off` can be latent, and has a latent fault probability of `1.e-6`. Notice that Common Causes cannot be declared to be latent.

7.4 TFPG Analysis

7.4.1 Associations file

A TFPG associations file has been created depending on the description contained at the beginning of Chapter 7. Here an extract of the complete example stored in file `examples/fe/triple_modular_generator/tfpg/SC_TMG.axml` is shown.

```
<associations>
  <failureModes>
    <!-- generator: stuck off -->
    <assoc id="G1_stuck_off" expr="SC.G1.Gen_StuckOff.mode_is_stuckAt_Off"/>
    <!-- circuit breakers: stuck open -->
    <assoc id="GB1_stuck_open"
           expr="SC.GB1.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Open"/>
    <assoc id="BB1_stuck_open"
           expr="SC.BB1.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Open"/>
```

```
    <!-- circuit breakers: stuck closed -->
    <assoc id="GB1_stuck_closed"
           expr="SC.GB1.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Closed"/>
    <assoc id="BB1_stuck_closed"
           expr="SC.BB1.Switch_StuckClosed_StuckOpen.mode_is_stuckAt_Closed"/>
  </failureModes>
  <monitoredDiscrepancies>
    <!-- triple power generation: three generators are on -->
    <assoc id="TriplePowerUsage" expr="count(SC.G1.is_on, SC.G2.is_on, SC.G3.is_on) = 3"/>
  </monitoredDiscrepancies>
  <unmonitoredDiscrepancies>
    <assoc id="B1_broken" expr="SC.B1.is_broken"/>
    <assoc id="InconsPowerUsage"
           expr="count(SC.G1.is_on, SC.G2.is_on, SC.G3.is_on) &gt; 0 &amp;
                 count(SC.B1.is_powered, SC.B2.is_powered, SC.B3.is_powered) &lt; 3"/>
  </unmonitoredDiscrepancies>
  <tfpgModes>
    <assoc id="R1"
           expr="(SC.GB1.is_open &amp; SC.GB2.is_closed &amp;
                  SC.GB3.is_closed &amp; SC.BB1.is_open &amp;
                  SC.BB2.is_closed &amp; SC.BB3.is_open)"/>
  </tfpgModes>
</associations>
```

For each `Generator` a failure mode `stuck_off` is created to represent the scenario in which a `Generator` breaks and propagates a failure through event `fev_stuck_at_off`. Additionally, two failure modes are added for each `Circuit Breaker` to depict the two admitted failure events, `fev_stuck_at_closed` and `fev_stuck_at_open`.

The discrepancy `TriplePowerUsage` is used to monitor the situation in which all the generators are powered on, while other discrepancies are included to check whether a bus is broken (e.g. `B1_broken`) or whether the power usage is inconsistent, i.e. some generator is working but not all buses are powered. These are some off-nominal conditions according to the system requirements described in Section 7.1.3.

Finally, the system mode `R1` is used to denote a specific switching configuration of the circuit breakers.

7.4.2 Synthesis

The associations file described in the previous section can be used to synthesize a TFPG for the running example. To do that, the SMV model `extended_SC_TMG.smv` previously generated using the script `extend_model.py` needs to be used. The following command can be run for TFPG synthesis:

```
$> cd examples/fe/triple_modular_generator/tfpg
$> python ../../../../scripts/synthesize_tfpg.py -a \
      SC_TMG.axml -m ../out/extended_SC_TMG.smv \
      -o SC_TMG_synthesized.txml --engine ic3 -b -B -c
```

For a complete description of script arguments see Section C.7.3. Depending on the model at hand, some engine settings might give better performance.

Result of synthesis

Here we show a possible result of a synthesis run. Any problems such as isolated failure mode nodes are reported at this output level.

```
tfpg synthesis > minimal cut set analysis
  --> Failure modes list dumped to out/extended_SC_TMG_sa_fm_list.xml
  --> Extended model dumped to 'out/extended_SC_TMG_synth_extended.smv'
  discrepancy 1/2 (TriplePowerUsage)
  discrepancy 2/2 (InconsPowerUsage)
tfpg synthesis > validating node names and cut sets
tfpg synthesis > processing unreachable discrepancies
tfpg synthesis > processing failure mode nodes
  failure mode 'GB1_stuck_open' has no effect on discrepancies within the analysis bound.
  failure mode 'GB2_stuck_open' has no effect on discrepancies within the analysis bound.
  failure mode 'GB3_stuck_open' has no effect on discrepancies within the analysis bound.
tfpg synthesis > checking for independent discrepancies
tfpg synthesis > checking for correlated discrepancies
tfpg synthesis > creating causality graph
tfpg synthesis > calling graph simplification routines
tfpg synthesis > writing result to file
  tfpg file: SC_TMG_synthesized.txml
```

The generated TFPG can then be shown with the TFPG Viewer:

```
$> cd examples/fe/triple_modular_generator/tfpg
$> python ../../../../scripts/view_tfpg.py -t SC_TMG_synthesized.txml
```

The TFPG Viewer opens and shows the TFPG:

The viewer offers the possibility of loading, saving and exporting a TFPG using the 'File' menu, while with the 'Edit' menu it is possible to edit the shown TFPG. In this case, an external editor with the textual representation of the TFPG is opened; to see the changes in the viewer the user must save and close the editor. Additionally, the toolbar allows to move, zoom and reset the shown TFPG.

7.4.3 Behavioral Validation

The synthesized TFPG can be validated using the `validate_tfpg_behavior.py` script. This can be done running the following command:

```
$> cd examples/fe/triple_modular_generator/tfpg
$> python ../../../../scripts/validate_tfpg_behavior.py -a \
      SC_TMG.axml -m ../out/extended_SC_TMG.smv -k 20 -b \
      --tfpg-file SC_TMG_synthesized.txml
```

Result of behavioral validation

An example output for behavioral validation is shown as follows.

```
  > extended smv model created (.../out/extended_SC_TMG_bv_extended_completeness.smv)
  > 5 proof obligations generated (.../out/extended_SC_TMG_proof_obligations_completeness.txt)

  > checking proof obligation 1/5 ... satisfied (within bound)
  > checking proof obligation 2/5 ... satisfied (within bound)
  > checking proof obligation 3/5 ... satisfied (within bound)
  > checking proof obligation 4/5 ... satisfied (within bound)
  > checking proof obligation 5/5 ... satisfied (within bound)

--------------------------------------------------------------------------------
Result: The TFPG is complete with respect to the model (within analysis bound).
--------------------------------------------------------------------------------
```

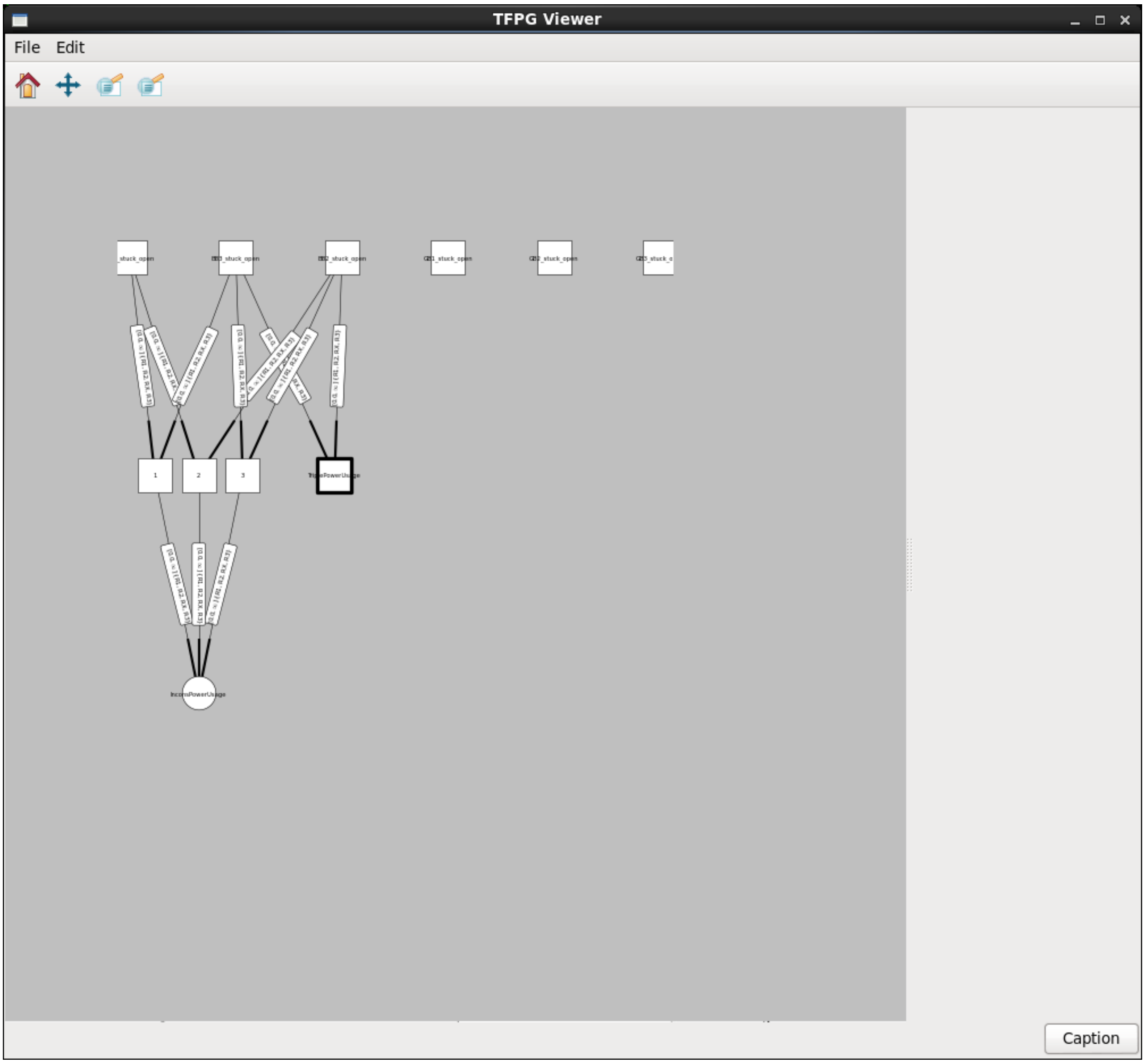


Figure 7.9: TFPG for file “SC_TMG_synthesized.txml”

7.4.4 Tightening

The synthesized TFPG can also be tightened using the `tighten_tfpg.py` script. This can be done running the following command:

```
$> cd examples/fe/triple_modular_generator/tfpg
$> python ../../../../scripts/tighten_tfpg.py -a SC_TMG.axml \
      -t SC_TMG_synthesized.txml -m ../out/extended_SC_TMG.smv
```

Note that the TMG use case is untimed, thus only mode labels will be tightened.

Result of tightening

An example output for tightening is shown as follows. Some output has been omitted for brevity. The user is continuously informed how many parameters remain to be tightened.

```
TFPG Tightening
```

```

  > skipping tightening of parameter 'tmin'
  > skipping tightening of parameter 'tmax'

  > initial completeness check
  > given TFPG is complete

  > tightening of modes
    44 parameters remaining ..
    43 parameters remaining .
    42 parameters remaining .
    [...]
    15 parameters remaining ..
    14 parameters remaining ....
    13 parameters remaining ....

  > instantiating tightened TFPG

------------------------------------------------
Result: The TFPG has been tightened.
Output file: out/SC_TMG_synthesized_tight.txml
------------------------------------------------
```

7.4.5 Statistics Information

The TFPG synthesized in the previous sections, can be used in order to retrieve some statistical information. Syntactical analysis can be performed by running:

```
$> cd example/fe/triple_module_generator/tfpg
$> python ../../../../scripts/stats_tfpg.py SC_TMG_synthesized.txml -s
```

This produces the following output, showing the number and types of nodes, edges and modes contained in the TFPG:

```
All Nodes: 11
FM Nodes: 6
AND Nodes: 4
OR Nodes: 1

Monitored Nodes: 1
Unmonitored Nodes: 4

Edges: 11

Modes: 4
```

7.4.6 Possibility, Necessity, Consistency and Activability

Possibility

Once we have synthesized our TFPG, we may want to check whether some traces are compatible or not with it.

The following trace says that, once the failure `BB3_stuck_open` is activated, as soon as the failure `BB2_stuck_open` is activated, so is `TriplePowerUsage`:

```
#BB3_stuck_open True
BB3_stuck_open 0
#BB2_stuck_open True
BB2_stuck_open 1
#TriplePowerUsage True
TriplePowerUsage 1
```

Indeed, running the following command:

```
$> cd example/fe/triple_module_generator/tfpg/smt
$> python ../../../../../scripts/check_tfpg.py --open-infinity
   --possibility --scenario scen_1.sc ../SC_TMG_synthesized.txml
```

we obtain a partial trace satisfying the scenario:

```
The scenario is possible!
A model is:
#intermediateNode_2 := True
#intermediateNode_3 := False
#intermediateNode_4 := False
BB3_stuck_open := 0.0
intermediateNode_4 := 3/2
#InconsPowerUsage := True
intermediateNode_2 := 1.0
intermediateNode_3 := 0.0
GB3_stuck_open := 0.0
#TriplePowerUsage := True
#BB1_stuck_open := False
BB1_stuck_open := 2.0
GB1_stuck_open := 0.0
GB2_stuck_open := 0.0
#GB2_stuck_open := False
TriplePowerUsage := 1.0
#BB3_stuck_open := True
BB2_stuck_open := 1.0
InconsPowerUsage := 1.0
#BB2_stuck_open := True
#GB3_stuck_open := False
#GB1_stuck_open := False
```

On the other side, if we run the same check with the following scenario, which states that `Triple_Power_Usage` can be activated before than `BB2_stuck_open`, we get that the specified scenario is not possible.

```
#BB3_stuck_open True
BB3_stuck_open 0
#BB2_stuck_open True
BB2_stuck_open 1
#TriplePowerUsage True
TriplePowerUsage 0
```

Necessity

Necessity is useful in case we want to check whether a particular scenario is implied by our TFPG; suppose to have the following simple scenario:

```
#TriplePowerUsage True
```

We are saying that the discrepancy `Triple_Power_Usage` is activated (no matter when); we can use the necessity check to verify that this is not implied by our TFPG, i.e. that the discrepancy is not always activated.

```
$> cd example/fe/triple_module_generator/tfpg/smt
$> python ../../../../../scripts/check_tfpg.py --open-infinity
   --necessity --scenario scen_2.sc ../SC_TMG_synthesized.txml
```

As expected, this is not the case:

```
The scenario is NOT necessary!
A counterexample scenario is:
#intermediateNode_2 := False
#intermediateNode_3 := False
```

```
#intermediateNode_4 := False
BB3_stuck_open := 0.0
intermediateNode_4 := −1.0
#InconsPowerUsage := False
intermediateNode_2 := −1.0
intermediateNode_3 := −1.0
GB3_stuck_open := 0.0
#TriplePowerUsage := False
#BB1_stuck_open := False
BB1_stuck_open := 0.0
GB1_stuck_open := 0.0
GB2_stuck_open := 0.0
#GB2_stuck_open := False
TriplePowerUsage := 1.0
#BB3_stuck_open := False
BB2_stuck_open := 0.0
InconsPowerUsage := −2.0
#BB2_stuck_open := False
#GB3_stuck_open := False
#GB1_stuck_open := False
```

Consistency

We can check whether our TFPG is consistent (i.e. there is at least a complete trace for it) by running the following command:

```
$> cd example/fe/triple_module_generator/tfpg/smt
$> python ../../../../../scripts/check_tfpg.py --open-infinity
   --consistency ../SC_TMG_synthesized.txml
$> The TFPG is consistent!
```

Activability

At last, we can check that all the nodes of our TFPG can be activated; this can be verified by executing the following command:

```
$> cd example/fe/triple_module_generator/tfpg/smt
$> python ../../../../../scripts/check_tfpg.py --open-infinity
   --activability ../SC_TMG_synthesized.txml
$> Checking if all nodes can be activated...
All nodes can be activated!
```

7.4.7 Diagnosis

Consider the following scenario, in which our observations refer to the failures `BB2`, `BB3`, `TriplePowerUsage` and to the mode `R3`.

```
#BB3_stuck_open True
#BB2_stuck_open True
#TriplePowerUsage True
#mode R3
```

We may want to enumerate all possible sets of failure modes compatible with the observations; this can be done using the following command:

```
$> cd example/fe/triple_module_generator/tfpg/smt
$> python ../../../../../scripts/compute_tfpg_diagnosis.py --open-infinity
     --diagnosability -d scen_3.sc ../SC_TMG_synthesized.txml
```

As a result, we obtain all the possible combinations of the remaining failures (`BB1`, `GB1`, `GB2` and `GB3`).

```
Getting all the diagnoses...
16 diagnoses found.
------
#BB3_stuck_open := True
#mode := R3
#GB3_stuck_open := False
#GB2_stuck_open := False
#GB1_stuck_open := False
#BB2_stuck_open := True
#BB1_stuck_open := False
------
#BB3_stuck_open := True
#mode := R3
#GB3_stuck_open := False
#GB2_stuck_open := False
#GB1_stuck_open := True
#BB2_stuck_open := True
#BB1_stuck_open := False
------
...
```

This corresponds to what we would expect, because in our synthesized TFPG the mode `R3` is enabled in all edges.

If we manually edit our TFPG by removing `R3` from the modes in the edges having destination `TriplePowerUsage`, we will se that no diagnosis is found, as in this case it is impossible to activate `TriplePowerUsage` remaining in mode `R3`.

7.4.8 Refinement

Suppose now that we want to restrict the set of possible behaviors of our synthesized TFPG. We can create another TFPG (see:
file `examples/fe/triple_modular_generator/tfpg/smt/SC_TMG_refined.tfpg`) and check whether it is a refinement of the original one.

As we can see, the new TFPG is built on top of the synthesized one, with the following changes:

- intermediate nodes have been renamed
- max time for the `InconsPowerUsage` has been set to 1.0
- modes over edges have been changed

We can verify that this new tfpg is a refinement of the original synthesized one by running the following command:

```
$> cd example/fe/triple_module_generator/tfpg/smt
$> python ../../../../../scripts/check_tfpg_refinement.py -m mapping.txt
   -r ../SC_TMG_synthesized.txml --open-infinity SC_TMG_refined.tfpg
$> Checking the refinement...
The given TFPG is a refinement of original_tfpg
```

7.4.9 Filtering

Whereas refinement validates a manual manipulation of a given TFPG, xSAP also provides automatic means to manipulate TFPGs. In the running example, for instance, we might be interested only in the paths leading to the discrepancy *TriplePowerUsage*. A TFPG containing only those paths can be created as follows:

```
$> cd examples/fe/triple_modular_generator/tfpg
$> python ../../../../scripts/filter_tfpg.py -t SC_TMG_synthesized.txml \
      --filter-action focus --focus-nodes TriplePowerUsage
```

The command will give the following output:

```
TFPG Filtering> computing reachability set
TFPG Filtering> removing nodes from which focus nodes cannot be reached
TFPG Filtering> dumping result

------------------------------------------------------------------------
The filtered TPFG file has been saved at 'out/SC_TMG_synthesized_focused.txml'
------------------------------------------------------------------------
```

The resulting TFPG contains only the failure mode nodes *B2_stuck_open* and *BB3_stuck_open*, as well as the discrepancy *TriplePowerUsage*.

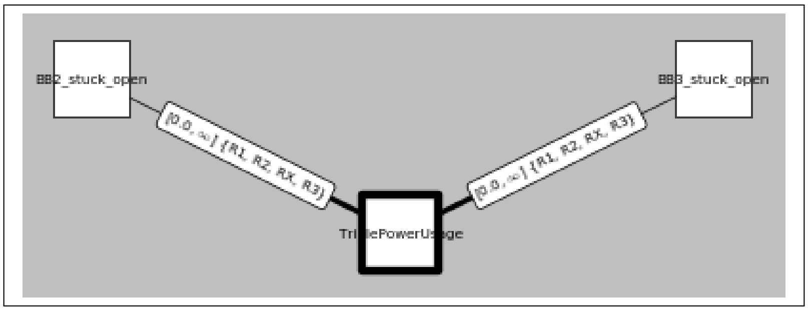


Figure 7.10: TFPG restricted to paths leading to TriplePowerUsage.

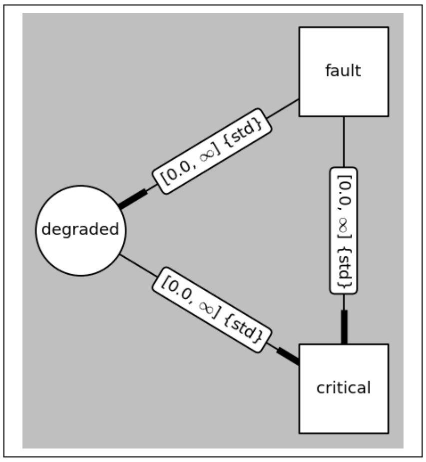


Figure 7.11: TFPG with a redundant edge.

Similarly, the script can be used to remove redundant edges on given TFPGs. For instance, consider the TFPG in Figure 7.11.

The edge from *fault* to *critical* is redundant. Invoking the script as follows will produce a new TFPG, where the redundant edge has been removed:

```
$> cd examples/fe/triple_modular_generator/tfpg
$> python ../../../../scripts/filter_tfpg.py -t SC_TMG_synthesized.txml \
      --filter-action simplify
```

As the simplification routines inside xSAP assume maximally permissive edges, this simplification functionality will only be executed if all edges have $t_{min} = 0$, $t_{max} = +\infty$ and the modes set to all modes known to the TFPG.

7.5 Fault Detection and Isolation

Fault Detection and Isolation (FDI) analysis is performed using the extended version of the Triple Module Generator model. In this model, the FDI model has been removed, as our goal is to synthesize it automatically from an FDI specification. The modified version can be found under file `examples/FDI/extended_SC_TMG_empty_controller.smv`.

In the next sections, we exemplify the different analyses separately.

7.5.1 Diagnosability analysis

As an example, we try to detect the generic fault for the generator G1. We expect that, knowing the received command and the state of G1, we should be able to successfully detect it. Hence, the observables file can be defined as follows:

```
SC.G1.state
CN.cmd_G1
```

The specific condition we want to diagnose is `SC.G1.Gen_StuckOff.mode != NOMINAL`, i.e. that G1 is failed at "StuckOff".

To run diagnosability, we can use the Python script provided by xSAP:

```
$> cd examples/FDI/diag
$> python ../../../scripts/check_diagnosability.py -m ../extended_SC_TMG_empty_controller.smv
      -a finite -c "SC.G1.Gen_StuckOff.mode != NOMINAL" -o G1_observables.obs
```

The check fails, and provides a pair of traces as counterexample; the fault occurs on the first trace but never on the second one, and yet both traces produce the same observations. We can see on the traces that after the fault the generator is never sent the command `cmd_on`, thus by observing only its state it is impossible to say whether the fault occurred or not. The traces can be rendered graphically using the script *view_trace.py*, as shown in appendix C.8.1.

We could however make an assumption on the environment or external controller that is not included in our model, and say that this command is sent periodically within a bound of 5 time units. Under this context restriction, the problem is indeed diagnosable within a delay bound of 5 time units.

In the previous example invoking the diagnosability script we specified the alarm condition directly on the command line. For the check with the context we now show how a specification file can be used instead (see Figure 7.12).

```
NAME: alarm_G1
CONDITION: SC.G1.Gen_StuckOff.mode != NOMINAL
TYPE: finite
CONTEXT: G F [0,5] CN.cmd_G1 = cmd_on
```

Figure 7.12: ASL file for diagnosability check with context.

Dignosability is then launched as follows.

```
$> python ../../../scripts/check_diagnosability.py -m ../extended_SC_TMG_empty_controller.smv
        -o G1_observables.obs --asl-file G1_with_context.asl
```

7.5.2 Minimum observables set analysis

We now want to check whether the observables we previously used are the minimum set needed to detect the fault of the generator G1; moreover, we want to check whether it is the only configuration which allows to detect it, or alternative configurations are possible. We increase for this analysis the set of observables that should be considered:

```
SC.G1.state
CN.cmd_G1
SC.G2.state
CN.cmd_G2
SC.G3.state
CN.cmd_G3
SC.GB1.state
CN.cmd_GB1
SC.GB2.state
CN.cmd_GB2
SC.GB3.state
CN.cmd_GB3
SC.BB1.state
CN.cmd_BB1
SC.BB2.state
CN.cmd_BB2
SC.BB3.state
CN.cmd_BB3
SC.B1_poweredby_G3_R
SC.B1_poweredby_G3_L
SC.B1_poweredby_G2_R
SC.B1_poweredby_G2_L
SC.B1_poweredby_G1_U
SC.B2_poweredby_G3_R
SC.B2_poweredby_G3_L
SC.B2_poweredby_G2_U
SC.B2_poweredby_G1_R
SC.B2_poweredby_G1_L
SC.B3_poweredby_G3_U
SC.B3_poweredby_G2_R
SC.B3_poweredby_G2_L
SC.B3_poweredby_G1_R
SC.B3_poweredby_G1_L
SC.B1_poweredby_G3
SC.B1_poweredby_G2
SC.B1_poweredby_G1
SC.B2_poweredby_G3
SC.B2_poweredby_G2
SC.B2_poweredby_G1
SC.B3_poweredby_G3
SC.B3_poweredby_G2
SC.B3_poweredby_G1
SC.B1.state
SC.B2.state
SC.B3.state
```

To run minimum observables set generation, we can run the following script provided by xSAP. The arguments are the same as for diagnosability checking, but now instead of verifying the observables we want to optimize them.

```
$> cd examples/FDI/diag
$> python ../../../scripts/minimize_observables.py -m ../extended_SC_TMG_empty_controller.smv
       -a bounded -d 5 -c "SC.G1.Gen_StuckOff.mode != NOMINAL"
       -x "G F [0,5] CN.cmd_G1 = cmd_on" -o full_observables.obs
```

The result consists of the following set:

```
1) -------------------------
   > SC.G1.state
   cost: 1
```

Note that the minimization procedure concludes that the command signal doesn't need to be observed, by assumption of context. The fault can thus be diagnosed by observing the state of generator G1, and triggering the alarm if it is off for more than 5 time units.

7.5.3 Synthesis of a diagnoser

A diagnoser for the generator G1 can be generated running the following xSAP commands. First we specify the definition of the observables and the alarms (associated with the fault we want to diagnose), and then we run the synthesis and write the generated model. The alarm specification file contains the following:

```
NAME: alarm_G1
CONDITION: SC.G1.Gen_StuckOff.mode != NOMINAL
TYPE: finite
CONTEXT: G F (CN.cmd_G1 = cmd_on)
```

In this particular case, we can omit the context and obtain the same diagnoser. However, the validation properties will show us that without context, we cannot always raise the alarm.

```
  $> cd examples/FDI/synth
  $> python ../../../scripts/synthesize_fd.py -m ../extended_SC_TMG_empty_controller.smv \
    -o G1_observables.obs -f G1.asl --out-file G1_synthesized_model.smv
```

where *G1_observables.obs* contains the observables of the system, and *G1.asl* contains the alarm specification.

The generated model, called *G1_synthesized_model.smv*, consists of the original model combined with the synthesized FDI module. An excerpt of the model is presented here:

```
MODULE __FD("CN.cmd_G1", "SC.G1.state")
    VAR
        __state : 1 .. 13;

    DEFINE
        Ualarm_G1 := ((__state = 4 | __state = 7) | __state = 11);
        Knalarm_G1 := (((__state = 2 | __state = 9) | __state = 5) | __state = 1);
        Kalarm_G1 := ((((__state = 8 | __state = 3) | __state = 10) | __state = 12) |
                        __state = 6);
```

The parameters of the *__FD* module are the observables we previously specified. The variable *state* is used to take into account all the possible states of the diagnoser; the system evolves depending on the possible combinations of the values of the observables.

Three DEFINE (*Ualarm_G1*, *Knalarm_G1* and *Kalarm_G1*) statements are added (notice that *alarm_G1* is the name we specified for the alarm) that define the states in which the fault expression is satisfied (*Kalarm_G1*), it is not satisfied (*Knalarm_G1*) or it is unknown (*Ualarm_G1*).

7.5.4 Effectiveness analysis

To validate the previous diagnoser (*G1_synthesized_model.smv*) we can model check the following three properties, that are automatically generated by the synthesis routines:

1. `LTLSPEC G (__myfdir.myfd.Knalarm_G1 ->`
   `!( O SC.G1.Gen_StuckOff.mode != NOMINAL))`
2. `LTLSPEC G (__myfdir.myfd.Kalarm_G1 ->`
   `O SC.G1.Gen_StuckOff.mode != NOMINAL)`
3. `LTLSPEC (G (SC.G1.Gen_StuckOff.mode != NOMINAL ->`
   `F __myfdir.myfd.Kalarm_G1))`

The first and second properties are *correctness* properties, and they should be valid for any synthesized diagnoser. The first property states that it is never the case that the fault occurred in the past, if the diagnoser knows for sure that it did not. The second property states that if the diagnoser knows for sure that the fault occurred, then it did indeed occur in the past. As expected, both properties hold.

The third property expresses *completeness* and holds if the given fault is diagnosable, not considering context. It encodes the fact that, if the system is in a state in which the mode is not nominal, eventually the corresponding fault alarm will be raised. This property doesn't hold in the model, since other behaviors outside the context are possible. However, when adding the context to the property, the proof obligation holds, indeed also within the time bound 5:

- `(G F [0,5] CN.cmd_G1 = cmd_on) ->`
  `G (SC.G1.Gen_StuckOff.mode != NOMINAL -> F [0,5] __myfdir.myfd.Kalarm_G1)`

Chapter 8

Conclusions and Future Directions

xSAP is a system that supports a formal, Model-Based Safety Assessment with along two main directions: *symbolic fault extension*, that allows the user to automatically obtain an model including the faulty behaviours from a source, nominal model; and procedures for *safety analysis*, such as FTA and FMEA, that allow the user to analyze the system under fault.

In the subsequent releases, the following extensions to xSAP will be considered. First, it will be instrumented to deal with asynchronous composition, and with continuous-time models [17]. Second, xSAP be extended to deal with fault-extension for a contract-based design flow, as described in [14]. Finally, xSAP will integrate capabilities for the analysis of reliability architectures [12, 13].

An extension to the probabilistic setting, as supported by the COMPASS toolset [11], is currently under consideration.

References

[1] Sherif Abdelwahed, Gabor Karsai, and Gautam Biswas. System diagnosis using hybrid failure propagation graphs. In *The 15th International Workshop on Principles of Diagnosis*. Citeseer, 2004.

[2] Sherif Abdelwahed, Gabor Karsai, Nagabhushan Mahadevan, and Stanley C Ofsthun. Practical implementation of diagnosis systems using timed failure propagation graph models. *Instrumentation and Measurement, IEEE Transactions on*, 58(2):240–247, 2009.

[3] Erika Ábrahám and Klaus Havelund, editors. *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*. Springer, 2014.

[4] Benjamin Bittner, Marco Bozzano, Alessandro Cimatti, and Gianni Zampedri. Automated Verification and Tightening of Failure Propagation Models. In *AAAI*, 2016.

[5] M. Bozzano, A. Cimatti, C. Mattarei, and A. Griggio. Efficient Anytime Techniques for Model-Based Safety Analysis. In *CAV*, pages 603–621, 2015.

[6] M. Bozzano, A. Cimatti, and F. Tapparo. Symbolic fault tree analysis for reactive systems. In *Automated Technology for Verification and Analysis (ATVA)*, volume 4762 of *LNCS*, pages 162–176. Springer, 2007.

[7] M. Bozzano, P. Munk, M. Schweizer, S. Tonetta, and V. Vozarova. Model-Based Safety Analysis of Mode Transitions. In *SafeComp*, pages X–X, 2020.

[8] Marco Bozzano, Alessandro Cimatti, Marco Gario, and Andrea Micheli. SMT-based validation of timed failure propagation graphs. In *AAAI*, 2015.

[9] Marco Bozzano, Alessandro Cimatti, Marco Gario, and Stefano Tonetta. Formal design of fault detection and identification components using temporal epistemic logic. In Ábrahám and Havelund [3], pages 326–340.

[10] Marco Bozzano, Alessandro Cimatti, Marco Gario, and Stefano Tonetta. Formal design of asynchronous FDI components using temporal epistemic logic. *Logical Methods in Computer Science*, 2015.

[11] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. Safety, dependability and performance analysis of extended aadl models. *Comput. J.*, 54(5):754–775, 2011.

[12] Marco Bozzano, Alessandro Cimatti, and Cristian Mattarei. Automated analysis of reliability architectures. In *ICECCS*, pages 198–207. IEEE, 2013.

[13] Marco Bozzano, Alessandro Cimatti, and Cristian Mattarei. Efficient analysis of reliability architectures via predicate abstraction. In Valeria Bertacco and Axel Legay, editors, *Haifa Verification Conference*, volume 8244 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2013.

[14] Marco Bozzano, Alessandro Cimatti, Cristian Mattarei, and Stefano Tonetta. Formal safety assessment via contract-based design. In *International Symposium on Automated Technology for Verification and Analysis*, 2014.

[15] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In Armin Biere and Roderick Bloem, editors, *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014.

[16] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Ic3 modulo theories via implicit predicate abstraction. In Ábrahám and Havelund [3], pages 46–61.

[17] The HyCOMP system. `https://es-static.fbk.eu/tools/hycomp/`.

[18] Gabor Karsai, Sherif Abdelwahed, and Gautam Biswas. Integrated diagnosis and control for hybrid dynamic systems. In *AIAA Guidance, AIAA Guidance, Navigation and Control Conference, Austin, Texas (August 2003)*, 2003.

[19] Amit Misra and J Sztipanovits. Diagnosability of dynamical systems. 1992.

[20] The nuXmv user manual. Available at `http://nuxmv.fbk.eu`.

[21] Space product assurance: Failure modes, effects (and criticality) analysis (FMEA/FMECA). ECSS Standard Q-ST-30-02C, European Cooperation for Space Standardization, March 2009.

[22] Space product assurance: Dependability. ECSS Standard Q-ST-30C, European Cooperation for Space Standardization, March 2009.

[23] Space product assurance: Fault tree analysis – adoption notice ECSS/IEC 61025. ECSS Standard Q-ST-40-12C, European Cooperation for Space Standardization, July 2008.

[24] Space product assurance: Safety. ECSS Standard Q-ST-40C, European Cooperation for Space Standardization, March 2009.

[25] SAE. ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996.

Appendix A

Installation

This chapter describes the necessary hardware/software configuration needed to run the xSAP tool and how to stay up to date with the latest updates.

A.1 Prerequisites

Since both Windows (64 bit and 32 bit) and Linux (64 bit) platforms are supported, and since Windows and Linux have very different packaging and installation procedures, and come with a largely different base of software, prerequisites are described separately in this section, for Linux and Windows systems.

Important: In directory `scripts` there is file `check_installation.py` which runs a set of tests to check installation requirments. Run it when done with the installation for a sanity check.

A.1.1 Platform-independent

SDEs All SDEs are Eclipse components which can be updated from this site: `https://es-static.fbk.eu/tools/devel_sde/`

Requisite: Eclipse Neon (`https://projects.eclipse.org/releases/neon`)

A.1.2 Microsoft Windows (64 bit and 32 bit)

nuXmv, ocra and xSAP OCRA, NUXMV and xSAP executables are statically linked, or when needed are shipped along with libraries they need.ma

However, a preprocessors like "cpp" and/or "m4" should be installed as separate packages. For "cpp", see for example GNU CPP at `https://gcc.gnu.org/`. For a binary executable see e.g. `http://tdm-gcc.tdragon.net/download` and select the version corresponding to the desired architecture (64 or 32 bit).

Scripts Scripts are all located in top-level directory file `scripts`.

To execute them, a Python 2.7.x interpreter is needed. It can be downloaded from here: `https://www.python.org/ftp/python/2.7.8/python-2.7.8.amd64.msi`

Viewers All requisites for running scripts, plus:

lxml Install selecting 64-bit package for the correct python version from `http://www.lfd.uci.edu/~gohlke/pythonlibs/#lxml`

gtk Version 2.22.1

1. Download bundle from here: `http://ftp.gnome.org/pub/gnome/binaries/win64/gtk+/2.22/gtk+-bundle_2.22.1-20101229_win64.zip`
2. Unpack e.g. in c:\opt\gtk-2.22.1
3. Put c:\opt\gtk-2.22.1\bin in system path
4. In a shell, try running gtk-demo to see if installation is OK

pygobject, py2cairo, pygtk Install selecting (64-bit or 32-bit) packages from `http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygtk` and `http://www.lfd.uci.edu/~gohlke/pythonlibs/#pycairo`

dateutil Version 2.2

Install selecting (64-bit or 32-bit) package for the correct python version from `http://www.lfd.uci.edu/~gohlke/pythonlibs/#python-dateutil`

numpy Version 1.8.1

Install selecting (64-bit or 32-bit) package for the correct python version from `http://www.lfd.uci.edu/~gohlke/pythonlibs/#numpy`

pyparsing Version 2.0.2

Install selecting (64-bit or 32-bit) package for the correct python version from `http://www.lfd.uci.edu/~gohlke/pythonlibs/#pyparsing`

matplotlib Version 1.3.1

Install selecting (64-bit or 32-bit) package from `http://www.lfd.uci.edu/~gohlke/pythonlibs/#matplotlib`

networkx Version 1.8.1 or later Install selecting (64-bit or 32-bit) package from `http://www.lfd.uci.edu/~gohlke/pythonlibs/#networkx`

pygraphviz Version 1.2 or later

1. Install graphviz-2.xx.msi (version 2.32 or later) from `http://www.graphviz.org/download/`
2. Install selecting (64-bit or 32-bit) package from: `http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygraphviz`

A.1.3 Linux 64 bit

nuXmv, ocra and xSAP OCRA, NUXMV and XSAP executables are statically linked.

Scripts Scripts are all located in top-level directory file `scripts`.

To execute them, a Python 2.7.x interpreter is needed. The Python interpreter should be already installed by default on any Linux distribution.

Should version 2.7.x be not installed, install the corresponding package (requires generally root priviledges).

Viewers All requisites for running scripts, plus:

lxml Version 3.2.3 or later. Install the corresponding package, or use pip.

gtk Version 2.22.1 or later, but stick with version 2.x. Install the corresponding package.

pygobject, py2cairo, pygtk Install corresponding packages.

matplotlib Version 1.3.1 or later

Install corresponding package.

networkx Version 1.8.1 or later

Install corresponding package.

graphviz Version 2.32 or later

Install corresponding package.

pygraphviz Version 1.2 or later

Install corresponding package.

goocanvas and pygoocanvas Version 1.x

Install corresponding packages.

Appendix B

Syntax Directed Editor

Syntactic Errors and Auto-completion The model viewer is able to detect and underline syntactic and semantic errors in the model and they can be viewed both in the related section of the code and in the "*Output Console*" which maintains also a history description. Another important feature of the model viewer is the auto-completion which aid the modeling by suggesting a set of reasonably possible keywords/words.

Appendix C

Script Guide

This section provides a reference for the scripts provided by xSAP.

C.1 Model Extender

The Model Extender takes as inputs:

- A nominal SMV model
- A Fault Extension Instruction file (`FEI`)

The Model Extender produces as output:

- A SMV model extended with fault behaviours
- A list of fault modes as XML file

Both outputs will be used in the Safety Analysis, while the extended SMV model can be used for model-checking. If the provided input contains errors, logging files will be produced.

The Model Extender is a combination of tools which process the input and perform the actual extension. The following script is available for this task.

```
python ../../scripts/extend_model.py -h
usage: extend_model.py [-h] [--xml-fei] [--verbose] [--disable-checks]
                       [-p PATH] [-d PATH] [--disable-cc] [--anonymize]
                       SMV-FILE FEI-FILE

Produces an extend smv file out of given nominal smv file and fei

positional arguments:
  SMV-FILE              The input nominal smv file name
  FEI-FILE              The input fei file name

optional arguments:
  -h, --help            show this help message and exit
  --xml-fei, -x         Process XML-format for input fei file
  --verbose, -v         Sets verbosity to high level
  --disable-checks, -c  Disable semantics checks when extending model (for
                        debugging)
  -p PATH, --path PATH  Path to the extension library (default:/hardmnt/mason0
                        /sra/bozzano/SMV/ESTools/xSAP/data/fm_library)
  -d PATH, --outdir PATH
                        Output directory, where all generated file should be
                        put into (default:out)
  --disable-cc, -C      Disable generation of common cause encoding when
                        extending model (for debugging)
  --anonymize, -A       Anonymize the generated extended model
```

All options are not mandatory, it can be run simply with:

```
$> python scripts/extend_model.py <nominal-model.smv> <fei.txt>
```

Additionally, option -v enables verbose messages to the user.

C.2 Fault Tree Analysis

Fault Tree analysis requires:

- A SMV model extended with fault behaviours
- A list of fault modes as XML file

It produces as output:

- A file containing the events of the generated fault tree
- A file containing the gates of the generated fault tree

If errors are encountered, logging files will be provided. The following script is available to call the FTA procedures.

```
python ../../scripts/compute_ft.py -h
usage: compute_ft.py [-h] [--smv-file SMV-FILE] [--fms-file FMS-FILE]
                     [--faults-bound BOUND] [--prop-index INDEX]
                     [--prop-name NAME] [--prop-text PROPERTY] [--verbose]
                     [--engine {bdd,bmc,bddbmc,msat,ic3,bmc_ic3}] [--dynamic]
                     [--gen-trace] [--bmc-length BMC_LENGTH] [--show]
                     [--probability] [--symbolic] [--quick] [--intermediate]
                     [-d PATH]
                     [--boolean-conversion-uses-predicate-normalization]

Produces an extend smv file out of given nominal smv file and fei

optional arguments:
  -h, --help            show this help message and exit
  --smv-file SMV-FILE   The input extended smv file name
  --fms-file FMS-FILE   The input fault mode xml file name
  --faults-bound BOUND, -N BOUND
                        Sets a bound to the maximum number of faults
  --prop-index INDEX, -n INDEX
                        Property index to be used as TLE
  --prop-name NAME, -P NAME
                        Property name to be used as TLE
  --prop-text PROPERTY, -p PROPERTY
                        Textual property to be used as TLE
  --verbose, -v         Sets verbosity to high level
  --engine {bdd,bmc,bddbmc,msat,ic3,bmc_ic3}, -E {bdd,bmc,bddbmc,msat,ic3,bmc_ic3}
                        Use given engine (default: bdd)
  --dynamic             Generates dynamic fault tree
  --gen-trace, -e       Generates xml trace from fault tree
  --bmc-length BMC_LENGTH, -k BMC_LENGTH
                        Specify BMC length (integer)
  --show, -s            Show the generated fault tree
  --probability         Computes probability when generating FT
  --symbolic, -S        Generates symbolic probability as well when computing
                        probability
  --quick, -Q           For quick computation, avoid ordering the FT and when
                        computing probability avoid computingprobability of
                        intermediate nodes. Use tospeedup computation.
  --intermediate, -I    Generates intermediate FTs for each layer
  -d PATH, --outdir PATH
                        Output directory, where all generated file should be
                        put into (default:out)
  --boolean-conversion-uses-predicate-normalization, -b
                        Enables predicate normalization during boolean
                        conversion
```

No option is required; if no smv model and fault modes files are provided, the script automatically looks for them in the `./out` directory. Files are selected only if there is no possible ambiguity, and choices are always reported by verbose messages. Additionally, option -v enables verbose messages to the user.

C.3 FMEA Table Analysis

FMEA Table analysis requires:

- A SMV model extended with fault behaviours
- A list of fault modes as XML file

It produces a textual file and an xml file containing the generated FMEA table. If errors are encountered, logging files will be provided. FMEA can be performed using the following script.

```
python ../../scripts/compute_fmea_table.py -h
usage: compute_fmea_table.py [-h] [--smv-file SMV-FILE] [--fms-file FMS-FILE]
                             [--prop-indices INDICES][--prop-names PROP-NAMES]
                             [--props-text PROPERTIES] [--verbose]
                             [--engine {bdd,bmc,msat}] [--dynamic] [--compact]
                             [--gen-trace] [--card CARD]
                             [--bmc-length BMC_LENGTH] [--show] [-d PATH]

Produces an extend smv file out of given nominal smv file and fei

optional arguments:
  -h, --help            show this help message and exit
  --smv-file SMV-FILE   The input extended smv file name
  --fms-file FMS-FILE   The input fault mode xml file name
  --prop-indices INDICES, -n INDICES
                        Property indices to be used as TLE (separated by ':',
                        ',', or spaces, ranges like '0-10' are allowed)
  --prop-names PROP-NAMES, -P PROP-NAMES
                        Property names to be used as TLE (separated by ':', or
                        ',')
  --props-text PROPERTY, -p PROPERTY
                        Textual property to be used as TLE(separated by ':' or
                        ',')
  --verbose, -v         Sets verbosity to high level
  --engine {bdd,bmc,msat}, -E {bdd,bmc,msat}
                        Use given engine (default: bdd)
  --dynamic             Generates dynamic fmea table
  --compact, -c         Generates compact fmea table
  --gen-trace, -e       Generates xml trace from fmea table
  --card CARD, -N CARD  Cut-Set cardinality (default: 1)
  --bmc-length BMC_LENGTH, -k BMC_LENGTH
                        Specify BMC length (integer)
  --show, -s            Show the generated fmea table
  -d PATH, --outdir PATH
                        Output directory, where all generated file should be
                        put into (default:out)
```

No option is required; if no SMV model and fault modes files are provided, the script automatically looks for them in the `./out` directory. Files are selected only if there is no possible ambiguity, and choices are always reported by verbose messages. Additionally, option -v enables verbose messages to the user.

C.4 MTCS Analysis

MTCS analysis requires:

- A SMV model extended with fault behaviours
- A list of events as XML file
- A list of expressions as command arguments

It produces as output:

- A file containing MTCS in XML format
- Optionally, a file containing MTCS in `dot` or `tex` format

```
usage: compute_mtcs.py [-h] [--smv-file SMV-FILE] [--fms-file FMS-FILE]
                       [--verbose] [--use-vars] [--paging]
                       [--visual-out FORMAT] [--layering] [--single-source]
                       --expressions ... [-d PATH]
                       [--boolean-conversion-uses-predicate-normalization]

Computes MTCS for given modes, smv file and fms file

optional arguments:
  -h, --help            show this help message and exit
  --smv-file SMV-FILE   The input extended smv file name
  --fms-file FMS-FILE   The input fault mode xml file name
  --verbose, -v         Sets verbosity to high level
  --use-vars, -V        Treats expressions as state mode variables
  --paging, -g          In visual output, print each transitions separately
  --visual-out FORMAT, -o FORMAT
                        Visual output format, either tex or dot
  --layering, -L        Turn off layering
  --single-source, -e   Compute transitions only from the first mode
  --expressions ...     List of expressions defining mode variables or modes
  -d PATH, --outdir PATH
                        Output directory, where all generated file should be
                        put into (default:out)
  --boolean-conversion-uses-predicate-normalization, -b
                        Enables predicate normalization during boolean
                        conversion
```

The only required option is the list of expressions. If no smv model and fault modes files are provided, the script automatically looks for them in the `./out` directory. Files are selected only if there is no possible ambiguity, and choices are always reported by verbose messages. Additionally, option -v enables verbose messages to the user.

C.5 Diagnosability

C.5.1 Diagnosability Analysis

Diagnosability analysis requires:

- An SMV model (extended with fault behaviours)
- A condition that needs to be diagnosed

The analysis answers positively if the condition is diagnosable, and provides a counterexample otherwise. If errors are encountered, logging files will be provided. Diagnosability analysis can be performed using the following script.

```
python scripts/check_diagnosability.py -h
usage: check_diagnosability.py [-h] [--engine {bdd,bmc,msat_bmc,ic3}]
                               [--bmc-length BMC_LEN] [--smv-file SMV]
                               [--diagnosis-condition DIAG_COND]
                               [--alarm-pattern {exact,bounded,finite}]
                               [--delay-bound DELAY_BOUND]
                               [--context-expression LTL_CONTEXT]
                               [--observables-file OBS_FILE]
                               [--asl-file ASL_FILE]
                               [--verbosity-level VERBOSE_LEVEL]

optional arguments:
  -h, --help            show this help message and exit
  --engine {bdd,bmc,msat_bmc,ic3}, -E {bdd,bmc,msat_bmc,ic3}
                        Use given engine (default: ic3)
  --bmc-length BMC_LEN, -k BMC_LEN
                        Maximum path length for BMC
  --smv-file SMV, -m SMV
                        SMV file
  --diagnosis-condition DIAG_COND, -c DIAG_COND
                        Diagnosis condition
  --alarm-pattern {exact,bounded,finite}, -a {exact,bounded,finite}
                        Alarm pattern (default: None)
  --delay-bound DELAY_BOUND, -d DELAY_BOUND
                        Alarm pattern delay bound
  --context-expression LTL_CONTEXT, -x LTL_CONTEXT
                        LTL context expression
  --observables-file OBS_FILE, -o OBS_FILE
                        File specifying observable variables
  --asl-file ASL_FILE, -f ASL_FILE
```

```
                         File containing the alarms specification
  --verbosity-level VERBOSE_LEVEL, -v VERBOSE_LEVEL
                         Sets the output verbosity level
```

C.5.2 Generation of Minimum Observables Set

Generation of minimum observables requires the same inputs as diagnosability analysis:

- An SMV model (extended with fault behaviours)
- A condition that needs to be diagnosed

The minimal sets of observables under which the condition is diagnosable will be returned. If errors are encountered, logging files will be provided. The analysis can be performed using the following script.

```
python ../../../scripts/minimize_observables.py -h
usage: minimize_observables.py [-h] [--engine {bmc,bmc_ic3}]
                               [--bmc-length BMC_LEN] [--smv-file SMV]
                               [--diagnosis-condition DIAG_COND]
                               [--alarm-pattern {exact,bounded,finite}]
                               [--delay-bound DELAY_BOUND]
                               [--context-expression LTL_CONTEXT]
                               [--observables-file OBS_FILE]
                               [--asl-file ASL_FILE]
                               [--verbosity-level VERBOSE_LEVEL]
                               [--no-stdout-print]

optional arguments:
  -h, --help            show this help message and exit
  --engine {bmc,bmc_ic3}, -E {bmc,bmc_ic3}
                        Use given engine (default: bmc)
  --bmc-length BMC_LEN, -k BMC_LEN
                        Maximum path length for BMC
  --smv-file SMV, -m SMV
                        SMV file
  --diagnosis-condition DIAG_COND, -c DIAG_COND
                        Diagnosis condition
  --alarm-pattern {exact,bounded,finite}, -a {exact,bounded,finite}
                        Alarm pattern (default: None)
  --delay-bound DELAY_BOUND, -d DELAY_BOUND
                        Alarm pattern delay bound
  --context-expression LTL_CONTEXT, -x LTL_CONTEXT
                        LTL context expression
  --observables-file OBS_FILE, -o OBS_FILE
                        File specifying observable variables
  --asl-file ASL_FILE, -f ASL_FILE
                        File containing the alarms specification
  --verbosity-level VERBOSE_LEVEL, -v VERBOSE_LEVEL
                        Sets the output verbosity level
  --no-stdout-print     Result is not printed to stdout.
```

C.6 FD Synthesis

FD synthesis requires:

- An SMV model (extended with fault behaviours)
- A set of observables variables
- An ASL specification

This provides in output an FD for the system that satisfies the ASL specification. If errors are encountered, logging files will be provided. FD Synthesis can be performed using the following script.

```
usage: synthesize_fd.py [-h] [--smv-file SMV] [--observables-file OBS_FILE]
                        [--asl-file ASL_FILE] [--stand-alone]
                        [--out-file TARGET_SMV]
                        [--verbosity-level VERBOSE_LEVEL]
                        [--composition-semantics COMPOSITION_SEMANTICS]
                        [--no-dag]

optional arguments:
  -h, --help            show this help message and exit
  --smv-file SMV, -m SMV
                        SMV file
  --observables-file OBS_FILE, -o OBS_FILE
                        File specifying observable variables
  --asl-file ASL_FILE, -f ASL_FILE
                        File specifying the alarm specification
  --stand-alone         Output only the FD (By default outputs the combination
                        of FD and plant)
  --out-file TARGET_SMV
                        Where to write the synthesized model
  --verbosity-level VERBOSE_LEVEL, -v VERBOSE_LEVEL
                        Sets the output verbosity level
  --composition-semantics COMPOSITION_SEMANTICS
                        Specify composition semantics:(s)ynchronous or
                        (a)synchronous
  --no-dag              Do not use the DAG printing of the transition
                        relation(This should be used only on small FD)
```

C.7 TFPG

C.7.1 Format Conversion

xSAP supports two formats for TFPGs: xml and textual. It is possible to switch between the two formats using the following script.

```
python ../../scripts/convert_tfpg_format.py
usage: convert_tfpg_format.py [-h] [--tfpg-file TFPG]
                              [--associations-file ASSOC]

Converts a TFPG (associations) XML file into textual format or viceversa

optional arguments:
  -h, --help            show this help message and exit
  --tfpg-file TFPG, -t TFPG
                        The TFPG file (can be in XML or human readable format)
  --associations-file ASSOC, -a ASSOC
                        TFPG SMV associations file
```

It is possible to convert both a TFPG or an associations file; the specific option is required.

C.7.2 TFPG Generation

TFPGs in xml format can be generated automatically. No options are required; if no filename is specified, a TFPG named `out.txml` is created.

```
python ../../scripts/generate_tfpg.py -h
usage: generate_tfpg.py [-h] [--output filename] [--min_lb n] [--max_lb n]
                        [--max_ub n] [--infinite_ub_probability n]
                        [--monitored_probability n]
                        [--and_over_or_probability n] [--modes n]
                        [--discrepancies n] [--edges n] [--failure_modes n]
                        [--graph type]

Random TFPG generator

optional arguments:
  -h, --help            show this help message and exit
  --output filename, -o filename
                        The file to write the TFPG to
  --min_lb n            The minimum lower bound in the TFPG
  --max_lb n            The maximum lower bound in the TFPG
  --max_ub n            The maximum upper bound in the TFPG
  --infinite_ub_probability n
                        The probability the maximum upper "_"bound in the TFPG
                        is infinite for each edge
  --monitored_probability n
                        The probability a discrepancy is "_"observable in the
                        TFPG
  --and_over_or_probability n
                        The probability a discrepancy is "_"an AND node (1 -
```

```
                         probability it is an OR node) in "_"the TFPG
  --modes n              The number of modes in the TFPG
  --discrepancies n      The number of discrepancies in the "_"TFPG
  --edges n              The number of edges in the TFPG
  --failure_modes n      The number of failure modes in the TFPG
  --graph type, -g type
                         The algorithm to be used for randomly generate a graph
```

C.7.3 TFPG Synthesis

TFPG synthesis is carried out specifying a SMV model extended with fault behaviours and an associations file. The synthesized TFPG is created in the chosen output directory. If no name is specified, the suffix `_synth_tfpg.txml` is added to the name of the SMV model. If errors are encountered, logging files will be provided. The following script can be used for this task.

```
python ../../scripts/synthesize_tfpg.py -h
usage: synthesize_tfpg.py [-h] [--associations-file ASSOC] [--smv-file SMV]
                          [--output-tfpg OUT_TFPG_FNAME]
                          [--engine {bmc,bdd,msat_bmc,sbmc,ic3,bmc_bdd,sbmc_bdd}]
                          [--force-boolean-ic3] [--bmc-length BMC_LEN]
                          [--dynamic]
                          [--boolean-conversion-uses-predicate-normalization]
                          [--disable-coi-reduction]
                          [--prune-unreachable-nodes]
                          [--verbosity-level VERBOSE_LEVEL] [--outdir PATH]

optional arguments:
  -h, --help            show this help message and exit
  --associations-file ASSOC, -a ASSOC
                        TFPG SMV associations file
  --smv-file SMV, -m SMV
                        SMV Model
  --output-tfpg OUT_TFPG_FNAME, -o OUT_TFPG_FNAME
                        Name of the synthesized TFPG
  --engine {bmc,bdd,msat_bmc,sbmc,ic3,bmc_bdd,sbmc_bdd}, -E {bmc,bdd,msat_bmc,sbmc,ic3,bmc_bdd,sbmc_bdd}
                        Use given engine (default: bmc)
  --force-boolean-ic3, -B
                        Force use of Boolean version of IC3 engine
  --bmc-length BMC_LEN, -k BMC_LEN
                        Maximum path length for BMC
  --dynamic, -D         Enables dynamic reordering of variables
  --boolean-conversion-uses-predicate-normalization, -b
                        Enables predicate normalization during boolean
                        conversion
  --disable-coi-reduction, -c
                        Disables TFPG graph simplification based on cone-of-influence
  --prune-unreachable-nodes
                        Enables a preprocessing step that removes unreachable
                        nodes before synthesis is started.
  --verbosity-level VERBOSE_LEVEL, -v VERBOSE_LEVEL
                        Sets the output verbosity level
  --outdir PATH, -O PATH
                        Output directory, where all generated file should be
                        put into (default: out)
```

If the required options are not specified, error messages are shown to the user. Option -v can be set to enable verbose messages to the user.

C.7.4 TFPG Behavioral Validation

TFPG behavioral validation is carried out specifying a SMV model extended with fault behaviours, an associations file and a TFPG. The result is printed on standard output and a file containing the proof obligations is generated. If errors are encountered, logging files will be provided. The following script can be used for the task.

```
python ../../scripts/validate_tfpg_behavior.py -h
usage: validate_tfpg_behavior.py [-h] [--tfpg-file TFPG]
                                 [--associations-file ASSOC] [--smv-file SMV]
                                 [--boolean-conversion-uses-predicate-normalization]
                                 [--dynamic-reordering]
                                 [--delta-t-var DELTA_T_VAR]
                                 [--bmc-length BMC_LEN]
                                 [--engine {msat_bmc,ic3}]
                                 [--property-to-validate {completeness,edge_tightness}]
                                 [--monolithic-check] [--generate-only]
```

```
                                    [--parametric-dump] [--outdir PATH]

optional arguments:
  -h, --help            show this help message and exit
  --tfpg-file TFPG, -t TFPG
                        The TFPG file (can be in XML or human readable format)
  --associations-file ASSOC, -a ASSOC
                        TFPG SMV associations file
  --smv-file SMV, -m SMV
                        SMV Model
  --boolean-conversion-uses-predicate-normalization, -b
                        Enables predicate normalization during boolean
                        conversion
  --dynamic-reordering, -D
                        Enables dynamic reordering of BDD variables
  --delta-t-var DELTA_T_VAR, -d DELTA_T_VAR
                        delta t variable name
  --bmc-length BMC_LEN, -k BMC_LEN
                        Maximum path length for BMC
  --engine {msat_bmc,ic3}, -E {msat_bmc,ic3}
                        Use given engine (default: msat_bmc)
  --property-to-validate {completeness,edge_tightness}, -p {completeness,edge_tightness}
                        Validate given property (default: completeness)
  --monolithic-check, -M
                        Check completeness proof obligations using a single
                        monolithic check
  --generate-only, -g   Only generate files for model checking
  --parametric-dump, -P
                        Dump parameterized completeness property
  --outdir PATH, -O PATH
                        Output directory, where all generated file should be
                        put into (default: out)
```

If the required options are not specified, error messages are shown to the user.

C.7.5 TFPG Tightening

TFPG tightening is carried out specifying a SMV model extended with fault behaviours, an associations file and a TFPG. The result is printed on standard output and a file containing the tightened TFPG is generated. If errors are encountered, logging files will be provided. The following script can be used for the task.

```
python ../../scripts/tighten_tfpg.py -h
usage: tighten_tfpg.py [-h] [--tfpg-file TFPG] [--associations-file ASSOC]
                       [--smv-file SMV] [--tags TAGS]
                       [--delta-t-var DELTA_T_VAR]
                       [--engine {conc,ia}] [--tmax-bound TMAX_BOUND]
                       [--bmc-length BMC_LEN] [--ic3-length IC3_LEN]
                       [--outdir PATH]

optional arguments:
  -h, --help            show this help message and exit
  --tfpg-file TFPG, -t TFPG
                        The TFPG file (can be in XML or human readable format)
  --associations-file ASSOC, -a ASSOC
                        TFPG SMV associations file
  --smv-file SMV, -m SMV
                        SMV Model
  --tags TAGS, -T TAGS  Parameters to be tightened (tmin/tmax/modes; semi-
                        colon separated)
  --delta-t-var DELTA_T_VAR, -d DELTA_T_VAR
                        delta t variable name
  --engine {conc,ia}, -E {conc,ia}
                        IC3 mode (default: conc)
  --tmax-bound TMAX_BOUND
                        Upper bound for tmax (for tightening of
                        tmax=infinity).
  --bmc-length BMC_LEN, -k BMC_LEN
                        Maximum path length for BMC
  --ic3-length IC3_LEN, -K IC3_LEN
                        Maximum number of frames for IC3
  --outdir PATH, -O PATH
                        Output directory, where all generated file should be
                        put into (default: out)
```

If the required options are not specified, error messages are shown to the user.

C.7.6 TFPG Effectiveness Validation

TFPG effectiveness validation is carried out specifying a SMV model extended with fault behaviours and a target failure modes set. If errors are encountered, logging files will be provided. The following script is available for this task.

```
python ../../scripts/validate_tfpg_effectiveness.py -h
usage: validate_tfpg_effectiveness.py [-h] [--tfpg-file TFPG]
                                      [--target-fm-set FM_SET]
                                      [--sampling-rate SAMPLING_RATE]
                                      [--target-system-mode TARGET_SYSTEM_MODE]
                                      [--use-bmc] [--bmc-length BMC_LENGTH]
                                      [--outdir PATH]

optional arguments:
  -h, --help            show this help message and exit
  --tfpg-file TFPG, -t TFPG
                        The TFPG file (can be in XML or human readable format)
  --target-fm-set FM_SET
                        The set of FMs that need to be diagnosed (as a group
                        colon-separated
  --sampling-rate SAMPLING_RATE
                        The interval at which the monitored system is
                        monitored
  --target-system-mode TARGET_SYSTEM_MODE
                        The system mode for which effectiveness should be
                        analyzed
  --use-bmc             Use bounded model checking
  --bmc-length BMC_LENGTH
                        Maximum path length for BMC
  --outdir PATH, -O PATH
                        Output directory, where all generated file should be
                        put into (default: out)
```

If the required options are not specified, error messages are shown to the user.

C.7.7 TFPG Statistics Information Extraction

Statistic information of a TFPG can be extracted specifying a valid TFPG file. The following script is available for this task.

```
python ../../scripts/stats_tfpg.py -h
usage: stats_tfpg.py [-h] [--syntax-only] inputTFPG

Statistics information on TFPG

positional arguments:
  inputTFPG           The TFPG model to use

optional arguments:
  -h, --help          show this help message and exit
  --syntax-only, -s   Perform only syntactical analysis
```

If the required options are not specified, error messages are shown to the user.

C.7.8 TFPG Properties Check

Check for possibility, necessity, consistency and activability of a scenario is carried out specifying the desired scenario and a TFPG. The following script is available for this task.

```
python ../../scripts/check_tfpg.py -h
usage: check_tfpg.py [-h] [--scenario scenario] [--possibility] [--necessity]
                     [--consistency] [--activability] [--all-modes]
                     [--open-infinity]
                     inputTFPG

TFPG Check (SMT): Checks multiple properties of the given TFPG.

positional arguments:
  inputTFPG            The TFPG model to use

optional arguments:
  -h, --help           show this help message and exit
  --scenario scenario  The scenario to be used for checking
  --possibility, -p    Checks the possibility of the given scenario in the given TFPG
  --necessity, -n      Checks the necessity of the given scenario in the given TFPG
  --consistency, -x    Checks if a scenario exists for the given TFPG
  --activability       Checks if all nodes can be activated
  --all-modes          Enumerates the modes compatible with the scenario.
  --open-infinity      Force the usage of Open Infinity semantics
```

If the required options are not specified, error messages are shown to the user.

C.7.9 TFPG Scenario Diagnosis

Diagnosis of a TFPG (and of a scenario) is carried out by specifying the TFPG (and the desired scenario). The following script is available for this task.

```
python ../../scripts/compute_tfpg_diagnosis.py -h
usage: compute_tfpg_diagnosis.py [-h] [--diagnosability] [--diagnose scenario]
                                 [--all-diag node] [--open-infinity]
                                 inputTFPG

TFPG Diagnosis (SMT): Diagnose the scenario for the given TFPG.

positional arguments:
  inputTFPG             The TFPG model to use

optional arguments:
  -h, --help            show this help message and exit
  --diagnosability      Checks if all failure modes are diagnosable
  --diagnose scenario, -d scenario
                        Enumerate the possible diagnoses for the given scenario
  --all-diag node, -a node
                        Checks if the given node appears in all the diagnoses
  --open-infinity       Force the usage of Open Infinity semantics
```

If the required options are not specified, error messages are shown to the user.

C.7.10 TFPG Refinement Check

TFPG refinement check is carried out by specifiying the original TFPG, the refined one and a mapping file. The following script is available for this task.

```
python ../../scripts/check_tfpg_refinement.py -h
usage: check_tfpg_refinement.py [-h] [--mapping mapping] [--refine original]
                                [--open-infinity]
                                inputTFPG

TFPG Refinement (SMT):  Checks if a inputTFPG is a refinement of the original TFPG.

positional arguments:
  inputTFPG             The TFPG model to use (TFPG 1)

optional arguments:
  -h, --help            show this help message and exit
  --mapping mapping, -m mapping
                        The refinement mapping
  --refine original, -r original
                       TFPG 2
  --open-infinity       Force the usage of Open Infinity semantics
```

If the required options are not specified, error messages are shown to the user.

C.7.11 TFPG Filtering

TFPG filtering is carried out on a given TFPG file, using the following script:

```
python ../../scripts/filter_tfpg.py -h
usage: filter_tfpg.py [-h] [--tfpg-file TFPG]
                      [--filter-action {simplify,focus}]
                      [--focus-nodes FOCUS_NODES] [--outdir PATH]

optional arguments:
  -h, --help            show this help message and exit
  --tfpg-file TFPG, -t TFPG
                        The TFPG file (can be in XML or human readable format)
  --filter-action {simplify,focus}, -F {simplify,focus}
                        'simplify': Basic simplification routines as executed in TFPG synthesis
                        are performed. Edges are assumed to be maximally permissive (tmin=0,
                        tmax=inf, modes=all), otherwise soundness is not guaranteed. Note that
                        all nodes in the input TFPG are preserved in the output TFPG.'focus':
                        All nodes and respective incoming edges from which the nodes specified
                        with '--focus-nodes' cannot be reached are dropped from the TFPG.
  --focus-nodes FOCUS_NODES
                        When selecting filter action 'focus', this option is used to indicate
                        the focus nodes to consider. Specify as colon-separated list of node names.
  --outdir PATH, -O PATH
                        Output directory, where all generated file should be put into (default: out)
```

C.8 Viewers

C.8.1 Trace viewer

The trace viewer can be used to display a trace generated by NUXMV or XSAP. It takes a trace file (XML format) and shows it. It allows also to specify filters which can be used to limit the symbols spaces and/or the steps which are show.

To run the Trace viewer, invoke script:

```
$> python ../../scripts/view_trace.py <trace-file.xml>
```

C.8.2 Fault Tree viewer

The fault tree viewer can be used to display a fault tree generated by XSAP. It takes a FT in FT-Plus format (two files for events and gates) or in XML format and shows it.

To run the Fault Tree viewer, invoke script:

```
$> python scripts/view_ft.py -h

usage: view_ft.py [-h] [--events-file EVENTS_FILE] [--gates-file GATES_FILE]
                  [--xml-file XML_FILE] [-d PATH] [-o OUTPUT_FILE]
                  [-f OUTPUT_FORMAT] [--verbose]

Opens the Fault Tree Viewerwith specified or found files

optional arguments:
  -h, --help            show this help message and exit
  --events-file EVENTS_FILE, -e EVENTS_FILE
                        The input events file name
  --gates-file GATES_FILE, -g GATES_FILE
                        The input gates file name
  --xml-file XML_FILE, -x XML_FILE
                        The input fault tree xml file name
  -d PATH, --outdir PATH
                        Input directory, where gates and events files are
                        searched when not specified (default:out)
  -o OUTPUT_FILE, --output-file OUTPUT_FILE
                        Output file where fault tree is dumped
  -f OUTPUT_FORMAT, --output-format OUTPUT_FORMAT
                        Output format for the fault tree (default: png)
  --verbose, -v         Sets verbosity to high level
```

Given a pair of events and gates files, (e.g. event.txt and gates.txt) it can show the fault tree with:

```
$> python scripts/view_ft.py -e events.txt -g gates.txt
```

The same can be obtained giving an xml representation of it (e.g. ft.xml):

```
$> python scripts/view_ft.py -x ft.xml
```

However, it can be handy to exploit option -d (or its default to `./out`) to automatically find fault tree generated by `scripts/compute_ft.py`. In this case, events and gates files are automatically selected only if there is no possible ambiguity, and choices are always reported by the verbose messages.

```
$> pwd
.../examples/fe/triple_modular_generator/

$> ls
out  SC_TMG.smv  SC_TMG.fei

$> python ../../../scripts/view_ft.py -v

INFO:Searching for suitable gates file
INFO:Found gates file 'out/extended_SC_TMGgates.txt'
INFO:Searching for suitable events file
INFO:Found events file 'out/extended_SC_TMGevents.txt'
```

This works only if `scripts/compute_ft.py` was called before invoking the ft viewer.

C.8.3 FMEA Table viewer

There is no viewer for FMEA at the moment. However, since FMEA is generated by xSAP as CSV file, text editors or spreadsheet programs like excel can be used to show it.

C.8.4 TFPG viewer

The TFPG viewer can be used to display a TFPG. It takes a TFPG in xml or textual format and shows it. The shown TFPG can also be moved, zoomed, edited and saved.

To run the TFPG viewer, invoke script:

```
$> python scripts/view_tfpg.py -h

usage: view_tfpg.py [-h] [--tfpg-file TFPG_FNAME] [--out-file OUT_FNAME]
                    [--output-format {pdf,png,dot}]

Opens the TFPG Viewer

optional arguments:
  -h, --help            show this help message and exit
  --tfpg-file TFPG_FNAME, -t TFPG_FNAME
                        The input TFPG file name
  --out-file OUT_FNAME, -o OUT_FNAME
                        The name of the output file used to dump the TFPG
  --output-format {pdf,png,dot}, -f {pdf,png,dot}
                        Output format (default: pdf)
```

It is possible to run the viewer in three ways:

1. with a TFPG file as argument; in this case, the selected TFPG is shown.
2. with a TFPG file and and output file as arguments; in this case, the corresponding TFPG is directly dumped to file
3. without parameters; in this case, an empty window is shown and it is possible to load the TFPG using the GUI.

Appendix D

Command Guide

This section provides a reference for the commands provided by xSAP.

We remark that xSAP also includes the commands of the nuXmv model checker. Please refer to the nuXmv documentation, available from the nuXmv home page (http://nuxmv.fbk.eu/)

D.1 Invoking xSAP

xSAP can be invoked from the command line in the following way:

```
$> <path_to_xsap>/xsap -int -sa_compass -sa_compass_task out/fms_SC_TMG.xml
  out/extended_SC_TMG.smv
```

where

- `out/fms_SC_TMG.xml` is the fault modes xml file;
- `out/extended_SC_TMG.smv` is the extended smv model.

The command-line options can be changed using environmental variables. This also makes it possible to change the name of the fault modes xml file within an xSAP session. For instance, the following is an example xSAP session.

```
$> <path_to_xsap>/xsap -int
xSAP > set input_file out/extended_SC_TMG.smv
xSAP > set sa_compass
xSAP > set sa_compass_task_file out/fms_SC_TMG.xml
```

D.2 Properties vs TLEs

Several commands accept as input properties and/or TLEs.
Properties describe a good behaviour of the system ("the airplane can always fly"), while a TLE represents a bad state, the TLE leading to a failure.

Properties are taken from the properties database and are read from the input SMV file or added at runtime with command `add_property`. When stored in the database, properties can be referred through their numeric ID (*index*) or optionally through their string name when available. When specifying a property in the database, both invariant and LTL specifications are allowed. Invariants accept the `next` operator, LTL specifications must be restricted to the fragment of LTL which can be translated to invariant with next. E.g. `G p`, `(G p) -> (G q)` are valid LTL properties for `Safety Assessment`. An example of an invalid LTL specification is `G (p -> F q)` which cannot be converted to an invariant.

Differently from properties, TLEs can be specified only as invariants specifications accepting the `next` operator.

The `Safety Assessment` commands transparently convert all legal LTL properties to an invariants, while negating them to obtain the corresponding TLEs, and base their analysis on the obtained TLEs.

D.3 Automated Fault extension

The fe_extend_module command performs automatic fault extension of the nominal model according to the previously loaded specification.

By using the information previously loaded with command fe_load_doc, the command performs the automatic fault extension of the currently loaded model and dumps the extended model to a new SMV file. Optionally, it creates also an XML file containing information about Fault Modes and Common Causes, with the possible associated information about fault probability.

Notice that at the moment the currently loaded FSM remains untouched, meaning that if you need to work on the extended model, you will have to reset and load it explicitly with read_model.

```
usage: fe_extend_module [-h] [-m fname] [-A] [-c] -o fname
  -h        Prints the command usage.
  -m fname  Dumps the fault mode predicates to the given file.
  -o fname  Dumps the extended HRC to the given file.
  -A        Anonymize the output.
  -c        Disable generation of Common Causes
```

The fe_load_doc command loads the fault extension specification XML file. Reads the given extension-file.xml containing the specification of the fault extension, and fills internal structures to prepare the extension. It performs syntactic and semantic checks, and report checking errors and warnings. The path to the directory containing file fe.dtd can be specified with option -p or with the environment variable XSAP_LIBRARY_PATH.

```
usage: fe_load_doc [-h] [-c] [-o fname] [-F xml|text] [-p path] -i fname
  -h        Prints this command usage
  -i fname  Loads the given XML extension file
  -c        Disables syntactic and semantic checks
  -o fname  Dumps errors and warnings to the specified output file
  -F text|xml   Format to be used when dumping errors and warnings [text]
  -p path   Sets the path where file 'fe.dtd' is located
```

D.4 Printing the Fault Variables

The `show_fault_variables` command lists the specified fault variables.

```
usage: show_fault_variables [-h] [-m| -o file] [-v]
   -h       Prints the command usage
   -m       Pipes output through the program specified by
            the "PAGER" shell variable if defined,
            else through UNIX "more"
   -o file  Writes the generated output to "file"
   -v       Prints verbosely
```

D.5 Computing Monotonic Fault Tree

The `compute_fault_tree` is the command to compute fault trees for monotonic systems, using the standard BDD-based engine.

```
usage: compute_fault_tree [-f] [-h] [-m| -o file] [-N NR_FAIL]
                          [-d] [-t [-Q]] [-e] [-p [-S]]
                          [-x "prefix_string"]
                          [-n index | -P name | "next-expr"]
   -h          Prints the command usage
   -m          Pipes output through the program specified by
               the "PAGER" shell variable if defined,
               else through UNIX "more"
   -o file     Writes the generated output to "file"
   -N NR_FAIL  Limit number of possible failures to NR_FAIL
   -d          Generates a dynamic fault tree
   -t          Generates fault tree
   -Q          Disables construction of ordered FT and computes probability
               only for the top level event (to speedup computation)
   -e          Prints counterexample traces
   -p          Computes probability
   -S          Computes probability also in symbolic form
               (python and octave/matlab)
   -x prefix   Prefixes generated file names with "prefix"
   -n index    Use given INVARSPEC or LTLSPEC property as !TLE
   -P name     Use given INVARSPEC or LTLSPEC property (from name) as !TLE
   next-expr   Use given expression as TLE

   Options -n, -P and "next-expr" are mutually exclusive.
   If none is specified, all INVARSPEC and LTLSPEC (which can be converted
   into INVARSPEC) properties will be used to generate
   multiple fault trees. In this case, the index of each property will
   appear in the prefix of generated files.

   Also notice that -n and -P take a property which is supposed to
   hold in the nominal model, i.e. that describes a good behaviour
   ("the airplane can always fly"). Instead the expression in "next-expr"
   represents a bad state, a failure TLE.
```

Example

```
xSAP > set input_file model.smv
xSAP > go
xSAP > compute_fault_tree -e -t "property-text"
```

The `compute_fault_tree_bmc` is the command to compute fault trees for monotonic systems, using the SAT-based engine.

```
usage: compute_fault_tree_bmc [-h] [-m| -o file] [-k length] [-l loopback]
                              [-T mcs] [-N NR_FAIL] [-d] [-t [-Q]] [-e]
                              [-p [-S]] [-x "prefix_string"]
                              [-n index | -P name | "next-expr"]
   -h          Prints the command usage
   -m          Pipes output through the program specified by
               the "PAGER" shell variable if defined,
               else through UNIX "more"
   -o file     Writes the generated output to "file"
   -k length   Set problem length to length
   -l loop     Set loopback value to loop
   -T mcs      Limit number of cut sets to be computed to mcs
   -N NR_FAIL  Limit number of possible failures to NR_FAIL
   -d          Generates a dynamic fault tree
   -t          Generates fault tree
   -Q          Disables construction of ordered FT and computes probability
               only for the top level event (to speedup computation)
   -e          Prints counterexample traces
   -p          Computes probability
   -S          Computes probability also in symbolic form
               (python and octave/matlab)
   -x prefix   Prefixes generated file names with "prefix"
   -n index    Use given INVARSPEC or LTLSPEC property as !TLE
   -P name     Use given INVARSPEC or LTLSPEC property (from name) as !TLE
   next-expr   Use given expression as TLE

   Options -n, -P and "next-expr" are mutually exclusive.
   If none is specified, all INVARSPEC and LTLSPEC (which can be converted
   into INVARSPEC) properties will be used to generate
   multiple fault trees. In this case, the index of each property will
   appear in the prefix of generated files.

   Also notice that -n and -P take a property which is supposed to
   hold in the nominal model, i.e. that describes a good behaviour
   ("the airplane can always fly"). Instead the expression in "next-expr"
   represents a bad state, a failure TLE.
```

Example

```
xSAP > set input_file model.smv
xSAP > go_bmc
xSAP > compute_fault_tree_bmc -e -t "property-text"
```

The `compute_fault_tree_bmc_inc` is the command to compute fault trees for monotonic systems, using the SAT-based engine and incremental verification. It requires an incremental SAT solver.

```
usage: compute_fault_tree_bmc_inc [-h] [-m| -o file] [-k length]
```

```
                              [-l loopback] [-T mcs] [-N NR_FAIL]
                              [-d] [-t [-Q]]
                              [-e] [-p [-S]] [-x "prefix_string"]
                              [-n index | -P name | "next-expr"]
  -h          Prints the command usage
  -m          Pipes output through the program specified by
              the "PAGER" shell variable if defined,
              else through UNIX "more"
  -o file     Writes the generated output to "file"
  -k length   Set problem length to length
  -l loop     Set loopback value to loop
  -T mcs      Limit number of cut sets to be computed to mcs
  -N NR_FAIL Limit number of possible failures to NR_FAIL
  -d          Generates a dynamic fault tree
  -t          Generates fault tree
  -Q          Disables construction of ordered FT and computes probability
              only for the top level event (to speedup computation)
  -e          Prints counterexample traces
  -p          Computes probability
  -S          Computes probability also in symbolic form
              (python and octave/matlab)
  -x prefix   Prefixes generated file names with "prefix"
  -n index    Use given INVARSPEC or LTLSPEC property as !TLE
  -P name     Use given INVARSPEC or LTLSPEC property (from name) as !TLE
  next-expr   Use given expression as TLE

  Options -n, -P and "next-expr" are mutually exclusive.
  If none is specified, all INVARSPEC and LTLSPEC (which can be converted
  into INVARSPEC) properties will be used to generate
  multiple fault trees. In this case, the index of each property will
  appear in the prefix of generated files.

  Also notice that -n and -P take a property which is supposed to
  hold in the nominal model, i.e. that describes a good behaviour
  ("the airplane can always fly"). Instead the expression in "next-expr"
  represents a bad state, a failure TLE.
```

Example

```
xSAP > set input_file model.smv
xSAP > go_bmc
xSAP > compute_fault_tree_bmc_inc -e -t "property-text"
```

The `compute_fault_tree_sbmc_inc` is the command to compute fault trees for monotonic systems, using the SAT-based engine, next bounded model checking and incremental verification. It requires an incremental SAT solver.

```
usage: compute_fault_tree_sbmc_inc [-h] [-m| -o file] [-k length]
                                   [-T mcs] [-N NR_FAIL] [-d] [-t [-Q]]
                                   [-e] [-p [-S]] [-V] [-B]
                                   [-c] [-x "prefix_string"]
                                   [-n index | -P name | "next-expr"]
   -h         Prints the command usage
   -m         Pipes output through the program specified by
              the "PAGER" shell variable if defined,
              else through UNIX "more"
   -o file    Writes the generated output to "file"
   -k length  Set problem length to length
   -T mcs     Limit number of cut sets to be computed to mcs
   -N NR_FAIL Limit number of possible failures to NR_FAIL
   -d         Generates a dynamic fault tree
   -t         Generates fault tree
   -Q         Disables construction of ordered FT and computes probability
              only for the top level event (to speedup computation)
   -e         Prints counterexample traces
   -p         Computes probability
   -S         Computes probability also in symbolic form
              (python and octave/matlab)
   -V         Does not perform virtual unrolling
   -B         Adds blocking clauses at all time steps
   -c         Performs completeness check
   -x prefix  Prefixes generated file names with "prefix"
   -n index   Use given INVARSPEC or LTLSPEC property as !TLE
   -P name    Use given INVARSPEC or LTLSPEC property (from name) as !TLE
   next-expr  Use given expression as TLE

   Options -n, -P and "next-expr" are mutually exclusive.
   If none is specified, all INVARSPEC and LTLSPEC (which can be converted
   into INVARSPEC) properties will be used to generate
   multiple fault trees. In this case, the index of each property will
   appear in the prefix of generated files.

   Also notice that -n and -P take a property which is supposed to
   hold in the nominal model, i.e. that describes a good behaviour
   ("the airplane can always fly"). Instead the expression in "next-expr"
   represents a bad state, a failure TLE.
```

Example

```
xSAP > set input_file model.smv
xSAP > go_bmc
xSAP > compute_fault_tree_sbmc_inc -e -t "property-text"
```

The `compute_fault_tree_bmc_bdd` is the command to compute fault trees for monotonic systems, using both the BDD-based and the SAT-based engine.

```
usage: compute_fault_tree_bmc_bdd [-h] [-m| -o file] [-k length]
                                  [-l loopback] [-T mcs]
                                  [-N NR_FAIL] [-d]
                                  [-t [-Q]] [-e] [-p [-S]]
                                  [-x "prefix_string"]
                                  [-n index | -P name | "next-expr"]
   -h          Prints the command usage
   -m          Pipes output through the program specified by
               the "PAGER" shell variable if defined,
               else through UNIX "more"
   -o file     Writes the generated output to "file"
   -k length   Set problem length to length
   -l loop     Set loopback value to loop
   -T mcs      Limit number of cut sets to be computed to mcs
   -N NR_FAIL  Limit number of possible failures to NR_FAIL
   -d          Generates a dynamic fault tree
   -t          Generates fault tree
   -Q          Disables construction of ordered FT and computes probability
               only for the top level event (to speedup computation)
   -e          Prints counterexample traces
   -p          Computes probability
   -S          Computes probability also in symbolic
               form (python and octave/matlab)
   -x prefix   Prefixes generated file names with "prefix"
   -n index    Use given INVARSPEC or LTLSPEC property as !TLE
   -P name     Use given INVARSPEC or LTLSPEC property (from name) as !TLE
   next-expr   Use given expression as TLE

   Options -n, -P and "next-expr" are mutually exclusive.
   If none is specified, all INVARSPEC and LTLSPEC (which can be converted
   into INVARSPEC) properties will be used to generate
   multiple fault trees. In this case, the index of each property will
   appear in the prefix of generated files.

   Also notice that -n and -P take a property which is supposed to
   hold in the nominal model, i.e. that describes a good behaviour
   ("the airplane can always fly"). Instead the expression in "next-expr"
   represents a bad state, a failure TLE.
```

The `compute_fault_tree_bmc_inc_bdd` is the command to compute fault trees for monotonic systems, using both the BDD-based and the SAT-based engine. It requires an incremental SAT solver.

```
usage: compute_fault_tree_bmc_inc_bdd [-h] [-m| -o file] [-k length]
                                      [-l loopback] [-T mcs] [-N NR_FAIL]
                                      [-d] [-t [-Q]] [-e]
                                      [-p [-S]] [-x "prefix_string"]
```

```
                                        [-n index | -P name | "next-expr"]
   -h         Prints the command usage
   -m         Pipes output through the program specified by
              the "PAGER" shell variable if defined,
              else through UNIX "more"
   -o   file  Writes the generated output to "file"
   -k length  Set problem length to length
   -l loop    Set loopback value to loop
   -T mcs     Limit number of cut sets to be computed to mcs
   -N NR_FAIL Limit number of possible failures to NR_FAIL
   -d         Generates a dynamic fault tree
   -t         Generates fault tree
   -Q         Disables construction of ordered FT and computes probability
              only for the top level event (to speedup computation)
   -e         Prints counterexample traces
   -p         Computes probability
   -S         Computes probability also in symbolic
              form (python and octave/matlab)
   -x prefix  Prefixes generated file names with "prefix"
   -n index   Use given INVARSPEC or LTLSPEC property as !TLE
   -P name    Use given INVARSPEC or LTLSPEC property (from name) as !TLE
  next-expr   Use given expression as TLE

   Options -n, -P and "next-expr" are mutually exclusive.
   If none is specified, all INVARSPEC and LTLSPEC (which can be converted
   into INVARSPEC) properties will be used to generate
   multiple fault trees. In this case, the index of each property will
   appear in the prefix of generated files.

   Also notice that -n and -P take a property which is supposed to
   hold in the nominal model, i.e. that describes a good behaviour
   ("the airplane can always fly"). Instead the expression in "next-expr"
   represents a bad state, a failure TLE.
```

Example

```
xSAP > set input_file model.smv
xSAP > go
xSAP > go_bmc
xSAP > compute_fault_tree_bmc_inc_bdd -e -t "property-text"
```

The `compute_fault_tree_sbmc_inc_bdd` is the command to compute fault trees for monotonic systems, using both the BDD-based and the SAT-based engine, simple bounded model checking and incremental verification. It requires an incremental SAT solver.

```
usage: compute_fault_tree_sbmc_inc_bdd [-h] [-m | -o file] [-k length]
                                        [-T mcs] [-N NR_FAIL] [-d] [-t [-Q]]
```

```
                                      [-e] [-p [-S]]
                                      [-V] [-B] [-c] [-x "prefix_string"]
                                      [-n index | -P name | "next-expr"]
   -h           Prints the command usage
   -m           Pipes output through the program specified by
                the "PAGER" shell variable if defined,
                else through UNIX "more"
   -o file      Writes the generated output to "file"
   -k length    Set problem length to length
   -T mcs       Limit number of cut sets to be computed to mcs
   -N NR_FAIL   Limit number of possible failures to NR_FAIL
   -d           Generates a dynamic fault tree
   -t           Generates fault tree
   -Q           Disables construction of ordered FT and computes probability
                only for the top level event (to speedup computation)
   -e           Prints counterexample traces
   -p           Computes probability
   -S           Computes probability also in symbolic form
                (python and octave/matlab)
   -V           Does not perform virtual unrolling
   -B           Adds blocking clauses at all time steps
   -c           Performs completeness check
   -x prefix    Prefixes generated file names with "prefix"
   -n index     Use given INVARSPEC or LTLSPEC property as !TLE
   -P name      Use given INVARSPEC or LTLSPEC property (from name) as !TLE
   next-expr    Use given expression as TLE

   Options -n, -P and "next-expr" are mutually exclusive.
   If none is specified, all INVARSPEC and LTLSPEC (which can be converted
   into INVARSPEC) properties will be used to generate
   multiple fault trees. In this case, the index of each property will
   appear in the prefix of generated files.

   Also notice that -n and -P take a property which is supposed to
   hold in the nominal model, i.e. that describes a good behaviour
   ("the airplane can always fly"). Instead the expression in "next-expr"
   represents a bad state, a failure TLE.
```

Example

```
xSAP > set input_file model.smv
xSAP > go
xSAP > go_bmc
xSAP > compute_fault_tree_sbmc_inc_bdd -e -t "property-text"
```

The `compute_fault_tree_msat_bmc` is the command to compute fault trees for monotonic systems, using the SMT-based engine. It requires an SMT solver.

```
usage: compute_fault_tree_msat_bmc [-h] [-m | -o file] [-k length]
                                   [-l loopback] [-T mcs] [-N NR_FAIL]
                                   [-t [-Q]] [-e] [-p [-S]] [-M]
                                   [-x "prefix_string"]
                                   [-n index | -P name | "next-expr"]
   -h          Prints the command usage
   -m          Pipes output through the program specified by
               the "PAGER" shell variable if defined,
               else through UNIX "more"
   -o file     Writes the generated output to "file"
   -k length   Set problem length to length
   -l loop     Set loopback value to loop
   -T mcs      Limit number of cut sets to be computed to mcs
   -N NR_FAIL Limit number of possible failures to NR_FAIL
   -t          Generates fault tree
   -Q          Disables construction of ordered FT and computes probability
               only for the top level event (to speedup computation)
   -M          Set the generation of monotonic cutsets to FALSE
   -e          Prints counterexample traces
   -p          Computes probability
   -S          Computes probability also in symbolic form
               (python and octave/matlab)
   -x prefix   Prefixes generated file names with "prefix"
   -n index    Use given INVARSPEC or LTLSPEC property as !TLE
   -P name     Use given INVARSPEC or LTLSPEC property (from name) as !TLE
   next-expr   Use given expression as TLE

   Options -n, -P and "next-expr" are mutually exclusive.
   If none is specified, all INVARSPEC and LTLSPEC (which can be converted
   into INVARSPEC) properties will be used to generate
   multiple fault trees. In this case, the index of each property will
   appear in the prefix of generated files.

   Also notice that -n and -P take a property which is supposed to
   hold in the nominal model, i.e. that describes a good behaviour
   ("the airplane can always fly"). Instead the expression in "next-expr"
   represents a bad state, a failure TLE.
```

Example

```
xSAP > set input_file model.smv
xSAP > go_msat
xSAP > compute_fault_tree_msat_bmc -e -t "property-text"
```

The `compute_fault_tree_param` is the command to compute fault trees for monotonic systems, using an engine based on IC3 and parameter synthesis, with a SAT or SMT back-end.

```
usage: compute_fault_tree_param [-h] [-m| -o file] [-t [-Q] [-I]] [-e]
                                [-p [-S]] [-N NR_FAIL]
                                [-x "prefix_string"] [-i] [-L][-k depth]
                                [-n index | -P name | "next-expr"]
   -h           Prints the command usage
   -m           Pipes output through the program specified by
                the "PAGER" shell variable if defined,
                 else through UNIX "more"
   -o file      Writes the generated output to "file"
   -t           Generates fault tree
   -Q           Disables construction of ordered FT and computes probability
                only for the top level event (to speedup computation)
   -e           Prints counterexample traces
   -p           Computes probability
   -S           Computes probability also in symbolic form
                (python and octave/matlab)
   -N NR_FAIL   Limit number of possible failures to NR_FAIL
   -x prefix    Prefixes generated file names with "prefix"
   -i           Forces the use of SMT for finite models
   -L           Disable layering
   -k depth     Use an initial BMC run up to the given depth
   -I           Generates intermediate fault trees for each layer
   -n index     Use given invar property as !TLE
   -P name      Use given invar property (from name) as !TLE
   next-expr    Use given expression as TLE

   Options -n, -P and "next-expr" are mutually exclusive.
   If none is specified, all INVARSPEC and LTLSPEC (which can be converted
   into INVARSPEC) properties will be used to generate
   multiple fault trees. In this case, the index of each property will
   appear in the prefix of generated files.

   Also notice that -n and -P take a property which is supposed to
   hold in the nominal model, i.e. that describes a good behaviour
   ("the airplane can fly"). Instead the expression in "next-expr"
   represents a bad state, a failure TLE.
```

Example

```
xSAP > set input_file model.smv
xSAP > go_bmc
xSAP > compute_fault_tree_param -e -t "property-text"
```

The `compute_fault_tree_param_klive` is the command to compute fault trees for monotonic systems, using the SAT-based engine and an invariant checking termination procedure based on IC3.

```
usage: compute_fault_tree_param_klive [-h] [-m| -o file] [-t [-Q] [-I]] [-L]
                                      [-A] [-e] [-p [-S]] [-N NR_FAIL]
                                      [-x "prefix_string"] [-i]
                                      [-n index | -P name | "ltl-expr"]
   -h         Prints the command usage
   -m         Pipes output through the program specified by
              the "PAGER" shell variable if defined,
              else through UNIX "more"
   -o file    Writes the generated output to "file"
   -t         Generates fault tree
   -Q         Disables construction of ordered FT and computes probability
              only for the top level event (to speedup computation)
   -L         Disable layering
   -A         Force the computing of upper and lower probability bounds
   -e         Prints counterexample traces
   -p         Computes probability
   -S         Computes probability also in symbolic
              form (python and octave/matlab)
   -N         NR_FAIL
   -x prefix Prefixes generated file names with "prefix"
   -i         Forces the use of SMT for finite models
   -I         Generates intermediate fault trees for each layer
   -n index   Use given ltl property as !TLE
   -P name    Use given ltl property (from name) as !TLE
   ltl-expr   Use given expression as TLE

   Options -n, -P and "ltl-expr" are mutually exclusive.
   Also notice that -n and -P take a property which is supposed to
   hold in the nominal model, i.e. that describes a good behaviour
   ("the airplane can always fly"). Instead the expression in "ltl-expr"
   represents a bad condition.
```

D.6 Computing Non Monotonic Fault Tree

The `compute_prime_implicants` computes prime implicants and generates a fault tree for a top level event given as a simple expression; non-failure variables are existentially quantified.

```
usage: compute_prime_implicants [-h] [-m| -o file] [-t] [-e] [-p [-S]]
                                [-x "prefix_string"]
                                [-n index | -P name | "next-expr"]
   -h         Prints the command usage
   -m         Pipes output through the program specified by
              the "PAGER" shell variable if defined,
              else through UNIX "more"
   -o file    Writes the generated output to "file"
   -t         Generates fault tree
   -e         Prints counterexample traces
   -p         Computes probability
   -S         Computes probability also in symbolic
              form (python and octave/matlab)
   -x prefix Prefixes generated file names with "prefix"
   -n index   Use given LTL property as !TLE
   -P name    Use given LTL property (from name) as !TLE
   next-expr Use given expression as TLE

   Options -n, -P and "next-expr" are mutually exclusive.
   If none is specified, all INVAR properties will be used to generate
   multiple fault trees. In this case, the index of each property will
   appear in the prefix of generated files.

   Also notice that -n and -P take a property which is supposed to
   hold in the nominal model, i.e. that describes a good behaviour
   ("the airplane can always fly"). Instead the expression in "next-expr"
   represents a bad state, a failure TLE.
```

Example

```
xSAP > set input_file model.smv
xSAP > go
xSAP > compute_prime_implicants -e -t "property-text"
```

The `compute_prime_implicants_param` computes prime implicants and generates a fault tree for a top level event given as a simple expression, using parameter synthesis.

```
usage: compute_prime_implicants_param [-h] [-m| -o file] [-t [-Q]]
                                      [-e] [-p [-S]]
                                      [-x "prefix_string"] [-i] [-L]
                                      [-n index | -P name | "next-expr"]
   -h         Prints the command usage
   -m         Pipes output through the program specified by
```

```
            the "PAGER" shell variable if defined,
            else through UNIX "more"
   -o file   Writes the generated output to "file"
   -t        Generates fault tree
   -Q        Disables construction of ordered FT and computes probability
             only for the top level event (to speedup computation)
   -e        Prints counterexample traces
   -p        Computes probability
   -S        Computes probability also in symbolic form
             (python and octave/matlab)
   -x prefix Prefixes generated file names with "prefix"
   -i        Forces the use of SMT for finite models
   -L        Disable layering
   -n index  Use given invar property as !TLE
   -P name   Use given invar property (from name) as !TLE
   next-expr Use given expression as TLE

   Options -n, -P and "simple-expr" are mutually exclusive.
   If none is specified, all invar properties will be used to generate
   multiple fault trees. In this case, the index of each property will
   appear in the prefix of generated files.

   Also notice that -n and -P take a property which is supposed to
   hold in the nominal model, i.e. that describes a good behaviour
   ("the airplane can fly"). Instead the expression in "simple-expr"
   represents a bad state, a failure TLE.
```

Example

```
xSAP > set input_file model.smv
xSAP > go_bmc
xSAP > compute_prime_implicants_param -e -t "property-text"
```

D.7 Computing FMEA Table

The `compute_fmea_table` command is the command to compute fmea tables, using the standard BDD-based engine.

```
usage: compute_fmea_table [-h] [-m| -o file] [-c] [-d] [-N NR_FAIL] [-t]
                          [-e] [-x "prefix_string"]
                          [-n <props> | -P <props> | "next-expr" ... "next-expr"]
   -h  Prints the command usage
   -m  Pipes output through the program specified by
       the "PAGER" shell variable if defined,
        else through UNIX "more"
   -c  Generates a compact FMEA table
   -d  Generates a dynamic FMEA table
   -N NR_FAIL  set number of failures to NR_FAIL (default: 1)
   -o file Writes the generated output to "file"
   -t  Generates fmea table
   -e  Prints counterexample traces
   -x prefix  Prefixes generated file names with "prefix"
   -n "props" A subset of INVARSPEC or LTLSPEC properties whose indices are given
  as argument. Indices are separated by comma ',' or colon ':'.
  Ranges are allowed where lower and upper bounds are separated
  by dash '-'. For example "1:3-6:8" for indices 1,3,4,5,6,8
   -P "props" A subset of INVARSPEC or LTLSPEC properties whose names are given
  as argument. Names are separated by comma ',' or colon ':'.

   Options -n, -P and "next-expr" ... "next-expr" are mutually exclusive.
   If none is specified, all INVARSPEC and LTLSPEC (which can be converted
   into INVARSPEC) properties will be used.

   Also notice that -n and -P take properties which are supposed to
   hold in the nominal model, i.e. that describe a good behaviour
   ("the airplane can always fly"). Instead the expressions in "next-expr"
   represent a bad state, a failure TLE.
```

Example

```
xSAP > set input_file model.smv
xSAP > go
xSAP > compute_fmea_table -t "property-text"
```

The `compute_fmea_table_bmc_inc` command is the command to compute FMEA tables, using the SAT-based engine and incremental verification. It requires an incremental SAT solver.

```
usage: compute_fmea_table_bmc_inc [-h] [-m| -o file] [-k length] [-l loopback]
                                  [-c] [-N NR_FAIL] [-t] [-e] [-x "prefix_string"]\
                                  [-n <props> | -P <props> | "next-expr" ... "next-expr
```

```
   -h  Prints the command usage
   -m  Pipes output through the program specified by
       the "PAGER" shell variable if defined,
        else through UNIX "more"
   -k length  set problem length to length
   -l loop  set loopback value to loop
   -c  Generates a compact FMEA table
   -N NR_FAIL  set number of failures to NR_FAIL (default: 1)
   -o file Writes the generated output to "file"
   -t  Generates fmea table
   -e  Prints counterexample traces
   -x prefix  Prefixes generated file names with "prefix"
   -n "props" A subset of INVARSPEC or LTLSPEC properties whose indices are given
  as argument. Indices are separated by comma ',' or colon ':'.
  Ranges are allowed where lower and upper bounds are separated
  by dash '-'. For example "1:3-6:8" for indices 1,3,4,5,6,8
   -P "props" A subset of INVARSPEC or LTLSPEC properties whose names are given
  as argument. Names are separated by comma ',' or colon ':'.

   Options -n, -P and "next-expr" ... "next-expr" are mutually exclusive.
   If none is specified, all INVARSPEC and LTLSPEC (which can be converted
   into INVARSPEC) properties will be used.

   Also notice that -n and -P take properties which are supposed to
   hold in the nominal model, i.e. that describe a good behaviour
   ("the airplane can always fly"). Instead the expressions in "next-expr"
   represent a bad state, a failure TLE.
```

Example

```
xSAP > set input_file model.smv
xSAP > go_bmc
xSAP > compute_fmea_table_bmc_inc -t "property-text"
```

The `compute_fmea_table_msat_bmc` command is the command to compute FMEA tables, using the SMT-based engine. It requires an SMT solver.

```
usage: compute_fmea_table_msat_bmc [-h] [-m| -o file] [-k length] [-l loopback]
                                   [-c] [-N NR_FAIL] [-t] [-e] [-x "prefix_string"]
                                   [-n <props> | -P <props> | "next-expr" ... "next-exp
   -h  Prints the command usage
   -m  Pipes output through the program specified by
       the "PAGER" shell variable if defined,
        else through UNIX "more"
   -k length  set problem length to length
   -l loop  set loopback value to loop
   -c  Generates a compact FMEA table
```

```
 -N NR_FAIL  set number of failures to NR_FAIL (default: 1)
 -o file Writes the generated output to "file"
 -t  Generates fmea table
 -e  Prints counterexample traces
 -x prefix  Prefixes generated file names with "prefix"
 -n "props" A subset of INVARSPEC or LTLSPEC properties whose indices are given
as argument. Indices are separated by comma ',' or colon ':'.
Ranges are allowed where lower and upper bounds are separated
by dash '-'. For example "1:3-6:8" for indices 1,3,4,5,6,8
 -P "props" A subset of INVARSPEC or LTLSPEC properties whose names are given
as argument. Names are separated by comma ',' or colon ':'.

 Options -n, -P and "next-expr" ... "next-expr" are mutually exclusive.
 If none is specified, all INVARSPEC and LTLSPEC (which can be converted
 into INVARSPEC) properties will be used.

 Also notice that -n and -P take properties which are supposed to
 hold in the nominal model, i.e. that describe a good behaviour
 ("the airplane can always fly"). Instead the expressions in "next-expr"
 represent a bad state, a failure TLE.
```

Example

```
xSAP > set input_file model.smv
xSAP > go_msat
xSAP > compute_fmea_table_msat_bmc -t "property-text"
```

D.8 Computing MTCS

```
usage: compute_mode_transition_cut_sets [-h] [-v] [-e] [-M] [-s] [-i] [-L]
                                        [-t] [-o format] [-g] [-x prefix]
                                        expr_1 ... expr_n
   -h  Prints the command usage
   -v  Use provided expressions as variables. The states are computed
       as all evaluations of the variables
   -e  Compute only transitions from the first mode to the others
   -M  Turn off monotonicity
   -s  Use strict MTCS computation
   -i  Forces the use of SMT for finite models
   -L  Disable layering
   -t  Print XML output in file
   -o format Print visual output in file (dot, tex)
   -g  Paging, print each mode transition in separate file
   -x prefix  Prefixes generated file names with "prefix"
```

Example

```
xSAP > set input_file model.smv
xSAP > go_bmc
xSAP > compute_mode_transition_cut_sets -t 'mode-expression'
                                           'mode-expression'
                                           'mode-expression'
```

D.9 Checking diagnosability

The build_twin_plant command is used to build the twin plant of a given model. Note that option "-p" is always required to be set, and kept only for backward-compatibility reasons.

```
usage: build_twin_plant -o <str> [-a] [-f] [-A] [-p] [-t <str>] [-h]
  -o  file containing list of observable variables
 [-a] asynchronous twinplant composition [option is deprecated] (default: false)
 [-f] configure twin plant for synthesis [option is deprecated] (c1/c2 specification)
 [-A] configure twin plant for synthesis (ASL specification)
 [-p] use history variables to encode twin plant
 [-t] name of delta_t variable [option is deprecated]
 [-h] prints command usage
```

Example

```
xSAP > set input_file examples/FDI/extended_SC_TMG_empty_controller.smv
xSAP > build_twin_plant -o examples/FDI/diag/G1_observables.obs -p
```

The diag_load_asl_spec command is used to load a diagnosis condition for diagnosability analysis.

```
usage: diag_load_asl_spec [-f <str>|-a <str> -c <str> -d <int> [-x <str>]] [-p] [-h]
    -f  file containing the list of ASL specifications
    -a  ASL alarm type (exact/bounded/finite)
    -c  diagnosis condition (LTL expression)
    -d  delay bound (for exact and bounded delay)
   [-x] context (LTL expression)
   [-p] remove existing properties from property manager
   [-h] prints command usage
```

Example

```
xSAP > set input_file examples/FDI/extended_SC_TMG_empty_controller.smv
xSAP > build_twin_plant -o examples/FDI/diag/G1_observables.obs
xSAP > diag_load_asl_spec -p -a finite -c "(CN.cmd_G1 = cmd_on &
    (SC.G1.Gen_StuckOff.mode != NOMINAL |
     SC.G1.Gen_StuckOff.event = stuckAt_Off#failure))"
```

Any available model-checking command can then be used to verify the generated diagnosability proof obligations. Multiple specifications can be loaded from a file using the option -f.

```
xSAP > set input_file examples/FDI/extended_SC_TMG_empty_controller.smv
xSAP > build_twin_plant -o examples/FDI/diag/G1_observables.obs -p
xSAP > diag_load_asl_spec -p -f examples/FDI/diag/G1.asl
```

D.10 Minimum observables set analysis

The `diag_optimize_observables_asl` command is used to synthesize a sets of observables guaranteeing diagnosability.

```
usage: diag_optimize_observables_asl [-f <str>|-a <str> -c <str> -d <int> [-x <str>]]
                                     [-e <str>] [-h]
    -f  file containing the list of ASL specifications
    -a  ASL alarm type (exact/bounded/finite)
    -c  diagnosis condition (LTL expression)
    -d  delay bound (for exact and bounded delay)
   [-x] context (LTL expression)
   [-e] engine (bmc/bmc_ic3; default: bmc_ic3)
   [-h] prints command usage
```

Similarly as for `diag_load_asl_spec` it is possible to specify either a single alarm specification, or multiple alarm specifications through the use of an ASL specification file. file. If a specification is given, the result is a sensor configuration that satisfies all alarm conditions at the same time.

```
xSAP > set input_file examples/FDI/extended_SC_TMG_empty_controller.smv
xSAP > build_twin_plant -o examples/FDI/diag/G1_observables.obs -p -A
xSAP > go_bmc
xSAP > diag_optimize_observables_asl -f examples/FDI/diag/G1.asl
```

D.11 Synthesizing FD components

The synth_asynchronous_composition_semantics environment variable sets the semantics of the composition between the FD and the system. If set to 0, it enforces the *synchronous* semantics; if set to 1, it enforces the asynchronous semantics. For SMV models, synchronous semantics should be used. See [10] for additional information.

Example

```
xSAP > set input_file examples/FDI/extended_SC_TMG_empty_controller.smv
xSAP > set synth_asynchronous_composition_semantics 1
xSAP > go
```

The synth_FD command is used to generate the FD component of a given model.

```
Performs the FD Synthesis taking into account the
observables list and the alarm specifications.
usage: synth_FD [-h] -o <file> -f <file>
        -o      file containing list of observable variables
        -f      file containing the ASL specification
        -h      prints the command usage.
```

Example

```
xSAP> set input_file examples/FDI/extended_SC_TMG_empty_controller.smv
xSAP> set synth_asynchronous_composition_semantics 0
xSAP> go
xSAP> synth_FD -o extended_SC_TMG.obs
               -f extended_SC_TMG.asl
```

With extended_SC_TMC.obs:

```
CN.cmd_G1
SC.G1.state
```

With extended_SC_TMC.asl:

```
NAME: FAULT
CONDITION: CN.cmd_G1 = cmd_on & (SC.G1.Gen_StuckOff.mode != NOMINAL &
           SC.G1.Gen_StuckOff.event = stuckAt_Off#failure)
TYPE: bounded
CONTEXT: GF (CN.cmd_G2 = cmd_on)
DELAY: 5
```

The synth_dump_fdir command is used to dump (to file) the generate model containing also the FD component.

```
usage: synth_dump_fdir [-o <file>] [-c] [-s]
        -o <file>       The file on which to print
        -c              Dumps the FDIR combined within the original FSM
        -s              Prints some additional statistics
        -v              Be verbose in the transition relation printing
```

Example

```
xSAP> set input_file examples/FDI/extended_SC_TMG_empty_controller.smv
xSAP> set synth_asynchronous_composition_semantics 0
xSAP> go
xSAP> synth_FD -o extended_SC_TMG.obs
               -f extended_SC_TMG.asl
xSAP> synth_dump_fdir -o G1_synthesized_model.smv -c
```