COMPASS

Correctness, Modeling, and Performance of Aerospace Systems

COMPASS User Manual

Version 3.0.1

Prepared by
Fondazione Bruno Kessler
RWTH Aachen University

Contents

Chapter 1

Introduction

This document provides the manual for the COMPASS (Correctness, Modeling, and Performance of Aerospace Systems) toolset. It is organized as follows:

- Chapter 2 lists the (abbreviated) terms that are applied in this document.
- Chapter 3 specifies the necessary hardware/software configuration needed to run the COMPASS toolset and the required installation steps.
- Chapter 4 describes the examples contained in the distribution.
- Chapter 5 explains the key features of the System-Level Integrated Modelling (SLIM). language that is employed for specifying systems.
- Chapter 6 describes the initial user action, the loading of SLIM files.
- Chapter 7 explains how to specify system properties.
- Chapter 8 describes the mission specification.
- Chapter 9 details the analysis features provided by the toolset.
- Chapter 10 describes how software maintenance and support is organized.

Chapter 2

Terminology

The following acronyms are used or are relevant in this document.

AADL	Architecture Analysis and Design Language
BDD	Binary Decision Diagram
BMC	Bounded Model Checking
CLI	Command-Line Interface
COMPASS	Correctness, Modeling, and Performance of Aerospace Systems
CSL	Continuous Stochastic Logic
CSSP	Catalogue of System and Software Properties
CTL	Computation Tree Logic
ECSS	European Cooperation for Space Standardization
EMA	Error Model Annex
ESA	European Space Agency
FDIR	Fault Detection, Identification, and Recovery
FMEA	Failure Modes and Effects Analysis
FTA	Fault Tree Analysis
GUI	Graphical User Interface
LTL	Linear Temporal Logic
NuSMV	New Symbolic Model Verifier
OCRA	Othello Contracts Refinement Analysis
RAMS	Reliability, Availability, Maintainability and Safety engineering
SAE	International Society of Automotive Engineers
SAT	Satisfiability
SLIM	System-Level Integrated Modeling
SMT	Satisfiability Modulo Theory

Chapter 3

Installation

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

3.1 Prerequisites

The COMPASS toolset is developed to target Ubuntu Linux 16.04. It is advised to have a fresh copy of this operating system installed before proceeding with the installation of the COMPASS toolset. In addition to a freshly installed Ubuntu 16.04, the following packages are also needed for running the toolset:

- python-glade2
- python-gtk2
- python-matplotlib
- python-networkx
- python-lxml
- python-nose
- python-pygraphviz
- python-setuptools
- python-tk
- python-tornado
- python-pygoocanvas

It is known that the toolset runs on other Linux distributions other than Ubuntu 16.04, as long as the above library versions and packages are installed.

3.2 COMPASS Toolset Packages

The COMPASS Toolset is made of:

- One main package called `compass-tools-<yyyymmdd>`. The package provides the core COMPASS application, distributed with an open source license which allow for redistributing it (with limitations, see the COMPASS General Public License for details.)
- One optional package called `compass-addons-noredist-<yyyymmdd>`. This package provides additional programs in binary format, which enable all the available features of the COMPASS Toolset. This package is distributed with a non open source license which does not allow for redistributing it, and adds some other restrictions, like for example commercial use. See the associated license for all details.

Notice that the additional package is not required, but some features will not be available if not installed. The COMPASS GUI will warn the user if it cannot be found when starting. In this manual, it is assumed that both the packages have been installed.

3.3 Obtaining a Copy of the Toolset

The most recent stable release of the COMPASS toolset is available on the COMPASS project website: `http://www.compass-toolset.org`. These releases are for ESA member states only. For non-ESA member states, please contact us using the information found on that website.

3.4 Installation of the COMPASS Toolset

After downloading the package `compass-tools-<yyyymmdd>.tar.gz` (and optionally the additional package `compass-addons-noredist-<yyyymmdd>.tar.gz`), you have to unpack it (them). From a Linux terminal:

```
$> cd <the download directory>
$> tar xvfz compass-tools-<yyymmdd>.tar.gz
```

If downloaded, unpack also the additional package, in the same directory where the former package was unpacked:

```
$> tar xvfz compass-addons-noredist-<yyyymmdd>.tar.gz
```

Note that these commands can be run as a normal user, there is no need for administrative rights to execute them.

At this point the downloaded package(s) can be removed.

3.5 Running the Toolset

There are two interfaces to the COMPASS Toolset, a command-line interface (CLI) and the graphical user interface (GUI). The command-line interface is primary used for regression testing purposes. For all other purposes, we recommend you to use the graphical user interface.

Graphical User Interface The executable of the GUI can be found in the `scripts`. The executable can be run as follows:

```
$> python scripts/compassw.py
```

A new window appears, which is the main window of the COMPASS toolset (see Figure 3.1).

The GUI executable accepts also a set of command line options, which are:

```
$> python scripts/compassw.py [--help|-h] \
                              [--wordsize|-w word_size] \
                              [--file|-f base_file_name] \
                              [--dir|-d out_dir_name] \
                              [--level|-l logging_sys_level] \
                              [--property|-p properties_file_name] \
                              [--mission|-m mission_file_name] \
                              [--tfpg|-t tfpg_file_name] \
                              [--slim_assoc|-a tfpg_assoc_file_name] \
                              [--finite_states] \
                              [slim-files...]
```

help,h prints the usage help

-w word_size sets the width of signed words used to represent `int` slim type data subcomponents. Default is 32 (bits).

-f base_file_name specifies the base file name of the files to be generated. Default value is the first input file without extension.

-d out_dir_name specifies the name of the directory in which to store output and logging files.

-l logging_sys_level sets the minimum system logging level.

-p properties_file_name loads the specified XML file containing the properties. The extension for properties file is `.propxml`

-m mission_file_name loads the specified XML file containing the mission specification. The extension for mission file is `.mxml`

-t tfpg_file_name loads the specified XML file containing the TFPG definition. The extension for tfpg file is `.txml`

-a tfpg_assoc_file_name loads the specified XML file containing the TFPG slim associations. The extension for tfpg slim associaions file is `.axml`

–**finite_states** maps integers to words.

slim-files... One or more slim file names separated by spaces, to be loaded initially.

Default values of these and further options can be set by editing entries in file `compass/options.py` before launching the GUI executable.

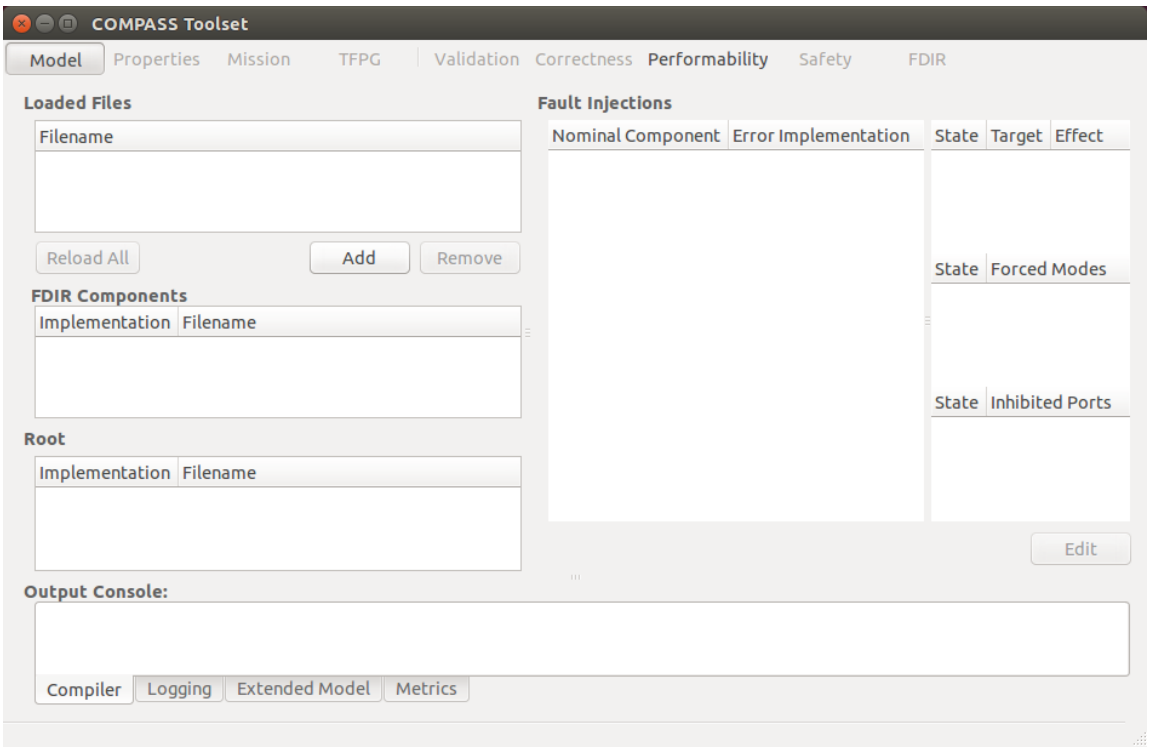


Figure 3.1: Main window of the COMPASS Toolset

Command Line Interface The scripts for the command-line interface are found in `scripts`. By running the script with `--help`, you will get a list of options. See an example below:

```
$> python scripts/evaluate_performability.py --help
Usage: evaluate_performability.py [options] slim-files

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
...
```

Chapter 4

Examples

This chapter briefly describes the examples that are contained in the COMPASS distribution, and their characteristics.

The examples can be found in `documentation/examples`.

4.1 Summary of Examples

We characterize the examples in terms of some metrics, such as size, complexity and the features they cover. This characterization may help the reader select a model for a specific purpose. The metrics are shown in the table below; for each example, we show its size (number of components, number of data ports), type (discrete, timed, infinite, continuous), features (presence of probabilistic error models, fault injections, contracts), and complexity[1]. Multiple labels refer to different models in the same example family.

Table 4.1: Characterization of examples.

Model	**Size**		**Type**	**Features**			**Complexity**
	#Comp	**#Data**		**Prob**	**F.Ext.**	**Contr.**	
adder	7-11	13-20	discrete, timed	N	Y	N	low, medium
battery_sensor	8-11	20-24	discrete, timed, continuous	N	Y	N	medium
blocks_world	5-11	10-14	timed, infinite	N	Y	N	medium, high
cruise	7	17	timed, infinite, continuous	N	N	N	medium
CSSP_EagleEye	4	10	discrete	N	N	Y	medium
engine	5	19	infinite	Y	N	N	medium

[1]Complexity characterization is provided as a simple qualitative performance indication, for user's guidance only. Performance may vary depending on the analysis that is being run.

Table 4.1: Characterization of examples.

Model	Size		Type	Features			Complexity
	#Comp	#Data		Prob	F.Ext.	Contr.	
features	1-5	0-5	discrete, timed, infinite	N	N	N	low
gps	1-7	2-6	timed, infinite	Y	Y	N	medium
new_semantics	3-5	0	discrete	N	N	N	low
power	1	4	continuous	Y	Y	N	low
sensor_filter	4-5	8-13	infinite, continuous	Y	Y	N	medium
smartgrid	1	28	infinite	N	N	N	medium
starlight	6	50	timed, infinite	N	N	Y	high
time_until	1-3	1-6	infinite	N	N	Y	low
VBE_Proc	3	4	timed	N	N	Y	low

4.2 Description of Examples

In this section we briefly describe the individual example families.

4.2.1 adder

The adder example is a simple example of an adder that computes the sum (modulo two) of some random input bits. It features the following components types: a component that generates the random inputs, a bit component that propagates its random input, an adder computing the sum of its input bits, and a scheduler that schedules the execution of the said components.

The example in available in the following variants:

- `adder_discrete`: discrete version with 2 bits
- `adder_discrete_3_bits`: discrete version with 3 bits
- `adder_timed`: timed version with 2 bits

The timed version encodes a timer, and explicitly forces the adder component to be triggered after a time delay (two similar versions are provided, one with time units and one without them).

The following faults are modeled: output of a bit may be inverted, output of the adder may be stuck at zero or one.

4.2.2 battery_sensor

The battery_sensor is a system featuring two generators, a pair of identical batteries and two sensors. The sensors provide some critical reading for the system, and it is therefore fundamental that, at any time, at least one of them provides a correct reading. In order to be able to power both sensors in case of failure of the generators, the sensors are connected to the batteries. The redundancy is intended to provide robustness in case of faults of the generators and/or of the batteries. Finally, the system may be re-configured dynamically – if a battery fails, the other battery may be connected to both sensors simultaneously.

The example is described in more detail in the COMPASS tutorial [7].

The example is available in the following variants:

- `system_discrete_simple`, `system_discrete` : discrete version
- `system_discrete_fdir`: discrete version with FDIR component
- `system_hybrid_1`, `system_hybrid_2`: hybrid versions
- `system_contracts`: version with contracts
- `system_reactive`: reactive version for probabilistic analyses

In the hybrid models, the charge level of the batteries is modeled with a hybrid dynamics.

The following faults are modeled: failure of a generator (no power), failure of a sensor (no reading); the batteries discharge whenever the corresponding generators fails.

4.2.3 blocks_world

The blocks_world example encodes a rail containing three sliding blocks. Each block has a letter, "A", "B", or "C" and the goal is to build a target word; in this case "ABC". To do so, we can control each rail independently, and build a suitable conformant plan.

The example is available in the following variants:

- `blocks_world`: timed version
- `blocks_world_fdir`: timed version with FDIR component

The variant with FDIR extends the basic model by combining it with an executor (which derives from the fdir) of the plan.

4.2.4 cruise

This example represents a redundant cruise control system for a car. It simply controls the speed of a car by accelerating, braking or staying. It is composed of two cruise control systems that are activated alternatively if a failure occurs in one of the two.

4.2.5 CSSP_EagleEye

This example represents a simplified model of a satellite taking picture of the Earth. Its purpose is to exemplify the usage of the CSSP properties.

4.2.6 engine

This example represents a 4-cylinder two-stroke car engine model. It models how fuel is transformed into power (in Newtons) and how that results in driven kilometers. One of a paired cylinder is required to work in order to have enough momentum to rotate the axis.

4.2.7 features

The features folder contains simple examples used to illustrate specific features of the SLIM language, such as clocks, event data ports, broadcast communication, tuple data types, constant, Zeno behaviors.

4.2.8 gps

The GPS example is a simplified representation of an on board GPS that, when activating, performs some initialization, then acquires the signal before becoming enabled.

The example is available in the following variants:

- `gps_clocks`: timed version
- `gps_delays`: version with integer variables to model delays
- `gps_fdir`: version with FDIR component
- `gps_lra`: version with transient failures
- `gps_nondet`: non-deterministic version
- `gps_timescale`: version with timescales

These model features several different variants of failure specifications. For all models, the GPS may lose the signal at any point in 98/255 time due to any type of failure: a transient failure, which recovers automatically, a permanent failure, which can never be recovered from, and a hot failure, which recovers after a system reset. If at any point in time the GPS loses signal, it will go into a failure state where it may retry to recover the signal, but remains in the failure mode if that is not possible (due to a permanent failure).

4.2.9 new_semantics

This folder contains a few examples that illustrate features covered by the new semantics of SLIM 3.0.

4.2.10 power

The power example is a simple system, in which two batteries are used to charge the voltage of a Power system. A battery may die, providing a zero output voltage.

4.2.11 sensorfilter

The sensorfilter example represents an acquisition system composed of sensors, filters and a monitor component. The system reads data from the sensors, filters it and forwards it via a value port, while the monitor observes the data values.

The example is available in the following variants:

- `sensorfilter`: version with integer variables
- `sensorfilter-deadlockfree`: version with integer variables, without deadlocks
- `sensorfilter-hybrid`: hybrid version

The following faults are modeled: wrong output of a sensor, wrong output of a filter.

4.2.12 smartgrid

The smartgrid model contains 2 prosumer (producer + consumer) components and one smart grid component. It describes the negotiation between the prosumers and the samrt grid for planning energy exchanges of the next day. The model features real variables.

4.2.13 starlight

The starlight example models the Starlight Interactive Link, which is a dispatching device developed by the Australian Defense Science and Technology Organization to allow users to esabilish simultaneous connections to high-level (classified) and low-level networks. The idea is that the device acts as a switch that the user can control to dispatch the keyboard output to either a high-level server or to a low-level server. The user can use the low-level server to browse the external world, send messages, or have data sent to the high-level server for later use. The model features contract specification.

4.2.14 time_until

This folder contains two simple examples illustrating the use of the **`time_until`** construct within contracts.

4.2.15 VBE_Proc

This model represents a voice service backend that is used as interface for secure communications. It uses two modules to implement a system that sends a ping and waits for response. The model features contract specification.

Chapter 5

The SLIM Language in a Nutshell

This section gives a short overview of the System-Level Integrated Modeling (SLIM) language. For the full specification refer to [12]. It has been designed to provide a cohesive and uniform approach to model heterogeneous systems, consisting of software (e.g., processes and threads) and hardware (e.g., processors and buses) components, and their interactions. Furthermore, it has been drafted with the following essential features in mind:

- Modeling both the system's nominal and non-nominal behavior. To this aim, SLIM provides primitives to describe software and hardware faults, error propagations (that is, turning fault occurrences into failure events), sporadic (transient) and permanent faults, and degraded modes of operation (by mapping failures from architectural to service level).
- Modeling (partial) observability and the associated observability requirements. These notions are essential to deal with diagnosability and Fault Detection, Isolation and Recovery (FDIR) analyses.
- Specifying timed and hybrid behavior. In particular, in order to analyse continuous physical systems such as mechanics and hydraulics, the SLIM language supports continuous real-valued variables with (linear) time-dependent dynamics.
- Modeling probabilistic aspects, such as random faults, repairs, and stochastic timing.

These features combined with a formal interpretation make SLIM suitable for specifying and reasoning about system properties from several perspectives, namely: functional correctness, in particular the case of degraded hardware operation; safety and dependability; diagnosability and FDIR; and performability, the system's performance under degraded operation.

A complete SLIM specification consists of three parts, namely a description of the nominal behavior, a description of the error behavior and a fault injection specification that describes how the error behavior influences the nominal behavior. These three parts are discussed in order below.

5.1 Nominal Behavior

A SLIM model is hierarchically organized into *components*, distinguished into software (processes, threads, data), hardware (processors, memories, devices, buses), and composite components (called *systems*). Components are defined by their *type* (specifying the functional

interfaces as seen by the environment) and their *implementation* (representing the internal structure).

Throughout the rest of this section, a power system will be used as an example. It comprises two batteries and a monitor that continuously checks the output voltages of the batteries, see also [1].

The monitor component checks the current voltage level and raises an alarm if it falls below a critical threshold of 4.5 [volts]. Its specification is shown in Figure 5.1. It consists of a component type `Monitor` and component implementation `Monitor.Imp`.

The component type describes the ports through which the component communicates. For example, the type interface of Figure 5.1 features two ports, namely an incoming data port `voltage` which is the input voltage to monitor, and an outgoing data port `alert` which indicates that the voltage is below the threshold.

The component implementation defines its subcomponents, their interaction through (event and data) port connections, the (physical) bindings at runtime, configurations and behavior. In the example of Figure 5.1, a single connection is defined, in this case a so-called data flow. A flow establishes a direct dependency between an outgoing data port of a component and (some of) its incoming data ports, meaning that a value update of one of the given incoming data ports immediately causes a corresponding update of the outgoing data port. In this example, the data port `alert` will be set to the value of the expression `(voltage < 4.5)`.

```
device Monitor
  features
    voltage: in data port real;
    alert: out data port bool;
end Monitor;

device implementation Monitor.Imp
  connections
    flow (voltage < 4.5) -> alert;
end Monitor.Imp;
```

Figure 5.1: Specification of the Monitor.

The power system itself is composed of two batteries and the monitor. An important feature is used here: mode configurations. In this example, two modes (`primary` and `backup`) define the possible configurations of the power system, see Figure 5.2. Furthermore, subcomponents are being defined, in this case a single monitor `mon` and two batteries `batt1` and `batt2`.

Mode transitions may give rise to modifications of a component's configuration: subcomponents can become (de-)activated and port connections can be (de-)established. This depends on the **`in modes`** clause, which can be declared along with port connections and subcomponents. In the example presented in Figure 5.2, the two instances of the battery device are being respectively active in the `primary` and the `backup` mode. The mode switch that initiates reconfiguration is triggered by an `empty` event arriving from the battery that is currently active.

A mode transition is of the form m `-[`e`]->` m'. It asserts that the component can evolve

```
system Power
  features
    alert: out data port bool;
end Power;

system implementation Power.Imp
  subcomponents
    batt1: device Battery.Imp {Accesses => (reference(myBus));}
             in modes (primary);
    batt2: device Battery.Imp {Accesses => (reference(myBus));}
             in modes (backup);
    mon: device Monitor.Imp {Accesses => (reference(myBus));};
    myBus: bus Bus;
  connections
    port batt1.voltage -> mon.voltage in modes (primary);
    port batt2.voltage -> mon.voltage in modes (backup);
    port mon.alert -> alert ;
    port mon.alert -> batt1.tryReset in modes (primary);
    port mon.alert -> batt2.tryReset in modes (backup);
  modes
    primary: initial mode;
    backup: mode;
    primary -[batt1.empty] -> backup;
    backup -[batt2.empty] -> primary;
end Power.Imp;

bus Bus
end Bus;
```

Figure 5.2: The Power System.

from mode m to mode m' on the occurrence of event e (the trigger event).The event e has to be reference an in event port of the component, or and out event port of a subcomponent.

In general, the mode transition system – basically a finite-state automaton – describes how the component evolves from mode to mode while receiving events.

The behavior of a component after a re-activation (that is, an activation following a previous de-activation) depends on the definition of its starting mode. If it is declared using the **initial** attribute (such as mode `primary` of the `Power` component in Figure 5.2), then *mode history* is supported, that is, after re-activation the component resumes its operation without changing its mode or the values of its data elements. In contrast, the **activation** attribute indicates that the component is to be reset to the starting mode, using the default values for its data elements.

Finally, the battery device is presented in Figure 5.3. An important feature presented here is the use of states, which describe the behavior of a component (as opposed to its configuration).

The behavior is described by states, possibly timed or hybrid, and transitions between

```
device Battery
  features
    empty: out event port;
    tryReset: in data port bool {Default => "false";};
    voltage: out data port real {Default => "6.0";};
end Battery;

device implementation Battery.Imp
  subcomponents
    energy : data continuous {Default => "1.0";};
  states
    charged: activation state
             while energy' = -0.02 and energy >= 0.2;
    depleted: state while energy' = -0.03 and energy >= 0.0;
  transitions
    charged -[then voltage := 2.0*energy+4.0]-> charged;
    charged -[reset when tryReset]-> charged;
    charged -[empty when energy <= 0.2]-> depleted;
    depleted -[then voltage := 2.0*energy+4.0]-> depleted;
    depleted -[reset when tryReset]-> depleted;
end Battery.Imp;
```

Figure 5.3: Specification of a Battery Component.

states, which can be spontaneous or triggered by any port event. For example, the implementation of Figure 5.3 specifies the battery to be in the `charged` state whenever activated, with an `energy` level of 100%. This level is continuously decreased by 2% (of the initial amount) per time unit (`energy'` denotes the first derivative of `energy`) until a threshold value of 20% is reached, upon which the battery changes to the `depleted` state. This state transition triggers the `empty` output event, and the loss rate of energy is increased to 3%. Moreover, the `voltage` value is regularly computed from the `energy` level (ranging between 6.0 and 4.0 [volts]) and made accessible to the environment via the corresponding outgoing data port. In addition, the battery listens to the `tryReset` port to decide when a **`reset`** operation should be performed in reaction to faulty behavior (see the description of error models below).

The behavior of the system also describes invariants on the values of data components (such as "`energy >= 0.2`" in state `charged`) restrict the residence time in a state. Trajectory equations (such as the one associated with `energy'`) specify how continuous variables evolve while residing in a state. This is akin to timed and hybrid automata [9]. Here we assume that all invariants are given by Boolean expressions over data subcomponents and constants where each arithmetic subexpression is linear. Moreover we constrain the derivatives occurring in trajectory equations to real constants, i.e., the evolution of continuous variables is described by linear functions.

It should also be noted that the `charged` state in Figure 5.3 is an **`activation`** state, which, like modes, indicates that upon (re-)activation the `Battery` component is reset, thus the model assumes that batteries will be recharged upon re-activation.

A state transition is of the form s **`-[`**e **`when`** g **`then`** f**`]->`** s'. It asserts that the component

can evolve from state s to state s' on the occurrence of event e (the trigger event) provided the guard g, a Boolean expression that may depend on the component's (discrete and continuous) data elements, holds. Here "data elements" refers to both (incoming and outgoing) data ports and data subcomponents of the respective component. On transiting, the effect f which may update data subcomponents or outgoing data ports (like `voltage`) is applied. The presence of event e, guard **when** g and effect **then** f is optional.

Important: States and modes cannot be mixed. Furthermore, it is not possible to define states for composite components (that is, any implementation that contains a non-data subcomponent).

5.1.1 Notes on Developing Timed Specifications

This section identifies three potential problems that have to be avoided when developing SLIM specifications of timed systems. They can all be illustrated by the simple component definition template that is shown in Figure 5.4.

```
system Timed
end Timed;

system implementation Timed.Imp
  subcomponents
    t0: data clock;
    t1: data clock;
  states
    m0: initial state while t0 <= b;
    m1: state while t1 <= d;
  transitions
    m0 -[when t0 >= a then t1 := 0]-> m1;
    m1 -[when t1 >= c then t0 := 0]-> m0;
end Timed.Imp;
```

Figure 5.4: A Timed Component Template.

In essence, it implements a mode transition loop of the form `m0` $\rightarrow$ `m1` $\rightarrow$ `m0`. This loop is governed by two timers, `t0` and `t1`, whose behavior is determined by two parameters each: the combination of the mode invariant of `m0` and of the outgoing transition guard expresses that this transition can (only) be taken in time interval $[a, b]$, while the conditions on `t1` impose that transition `m1` $\rightarrow$ `m0` is (only) enabled in interval $[c, d]$. Depending on the choice of values for those parameters, the following effects are possible.

Zeno Cycles

Description The notion of "Zeno behavior" refers to system computations involving an infinite number of discrete (i.e., mode transition) steps within a bounded period of time. This contradicts the natural assumption that only finitely many events can happen in a finite amount of time, which is reasonable for any realistic system. Zeno behavior can always be traced back to a cycle in the mode transition system of a component such that the delay to

take a full round of the cycle is zero, or can become arbitrarily small, so that the final sum of all delays can be finite. Such a cycle is named after Zeno, the Greek philosopher of around 500 BC., who was the author of a number of paradoxes (such as the paradox of Achilles and the tortoise).

Example. In Figure 5.4, Zeno behavior can occur when $a = c = 0$. This means that the cycle `m0` $\rightarrow$ `m1` $\rightarrow$ `m0` can be taken infinitely often within a bounded period of time (in fact, it can be executed without any time passing).

Avoidance. It must be guaranteed that the overall system does not give rise to an infinite sequence of mode transitions in which time converges to a bounded value. This can be ensured by requiring that, for every component of the system and on *each* cycle in the mode transition diagram of that component, there is a clock variable t that is reset, and that occurs in a transition guard of the form t `>` k or t `>=` k, where k is a positive constant. (These two actions do not necessarily have to occur in the same transition.) In the example this is ensured if $a > 0$ or $c > 0$.

Note that both conditions are mandatory. If the reset is absent, then once the guard is met it will stay enabled even if no time passes. Conversely, only resetting the clock does not require time to pass if the guard is absent. Also note that the clock reset/guard combination is in particular required for self loops, i.e., direct transitions from a mode to itself.

For a composite system of components, Zeno cycles are only possible if at least one of the components exhibits this behavior in isolation. In other words, the absence of Zeno cycles in each single component excludes Zeno cycles in the overall system behavior. In that sense, the property can be analyzed locally.

However note that this condition is sufficient but not necessary. Zeno behavior of the overall system can also be eliminated by synchronizing a component that exhibits Zeno cycles with one that does not, as in the example shown in Figure 5.5. Components without clocks (and with mode transition cycles), such as `Untimed` in this example, always exhibit Zeno behavior and therefore have to be synchronized to avoid this problem.

Timelocks

Description. In general, timelocks are caused by contradicting restrictions in the form of mode triggers, invariants or guards. In the simplest case, they can be localized in a single mode transition of a component. In more complicated settings, they are due to the synchronization between several components.

Example. The simplest form of timelock can be represented by a transition of the form `m -[when false]->n` (if no other outgoing transition is enabled in `m`). In Figure 5.4, a timelock occurs when $[a, b] = \emptyset$, e.g., when $a = 2$ and $b = 1$. The same is true when $[c, d] = \emptyset$.

Avoidance. The main problem with timelocks is that the corresponding analysis is non-compositional: the combination of two (or more) subsystems without timelocks can result in time-locking behavior, as shown in Figure 5.6. Here, both subsystems are free from timelocks. However after composition, the timing restrictions (output from `Timed1` in interval $[1, 2]$,

```
system Synchro
end Synchro;
system implementation Synchro.Imp
  subcomponents
    timed: system Timed {Accesses => (reference(mybus));};
    untimed: system Untimed {Accesses => (reference(mybus));};
    mybus: bus Bus;
  connections
    port timed.sync -> untimed.sync;
end Synchro.Imp;

system Timed
  features
    sync: out event port;
end Timed;
system implementation Timed.Imp
  subcomponents
    t: data clock;
  states
    m0: initial state while t <= 2;
  transitions
    m0 -[sync when t >= 1 then t := 0]-> m0;
end Timed.Imp;

system Untimed
  features
    sync: in event port;
end Untimed;
system implementation Untimed.Imp
  modes
    n0: initial mode;
    n0 -[sync]-> n0;
end Untimed.Imp;

bus Bus
end Bus;
```

Figure 5.5: Avoidance of Zeno Cycles by Synchronization.

input to `Timed2` in interval $[3, 4]$) exclude synchronization and thus cause a timelock. Note that untimed deadlocks can be considered as a special case of timelocks.

In summary, it is not easy to cope with timelocks. In fact, they can even be part of the expected system behavior. This applies in situations where one wants to show that under the given timing restrictions, the system activities cannot be scheduled in such a way that these restrictions are met.

```
system TimeLock
end TimeLock;
system implementation TimeLock.Imp
  subcomponents
    timed1: system Timed1 {Accesses => (reference(mybus));};
    timed2: system Timed2 {Accesses => (reference(mybus));};
    mybus: bus Bus;
  connections
    port timed1.sync -> timed2.sync;
end TimeLock.Imp;

system Timed1
  features
    sync: out event port;
end Timed1;
system implementation Timed1.Imp
  subcomponents
    t: data clock;
  states
    m0: initial state while t <= 2;
  transitions
    m0 -[sync when t >= 1 then t := 0]-> m0;
end Timed1.Imp;

system Timed2
  features
    sync: in event port;
end Timed2;
system implementation Timed2.Imp
  subcomponents
    t: data clock;
  states
    n0: initial state while t <= 4;
  transitions
    n0 -[sync when t >= 3 then t := 0]-> n0;
end Timed2.Imp;

bus Bus
end Bus;
```

Figure 5.6: A Time Lock Caused by Synchronization.

Time Divergence

Description. The notion of time divergence refers to the situation that there exists a computation trace on which a clock may grow arbitrarily.

Example. In Figure 5.4, time divergence occurs when both transitions are enabled infinitely often and when one of the timer resets is missing. This is the case, for example, if both the effect `t1 := 0` in the transition from `m0` to `m1` and the mode invariant `t1 <= d` of `m1` are removed from the specification.

Avoidance. Time divergence can easily be excluded by requiring for each component, say C, of the system that on every cycle in the mode transition system of C, every clock of C must be reset at least once.

Just like Zeno behavior and in contrast to timelocks, time divergence is compositional in the sense that the combination of subsystems that are not time diverging yields a system with the same property.

5.2 Error Behavior

Error models in SLIM are an extension to the specification of nominal models and are used to conduct safety and dependability analyses. For modularity, they are defined separately from nominal specifications. Akin to nominal models, an error model is defined by its type and its associated implementation.

An error model *type* defines an interface in terms of error states and (incoming and outgoing) error propagations. Error *states* are employed to represent the current configuration of the component with respect to the occurrence of errors. Error *propagations* are used to exchange error information between components.

An error model *implementation* provides the structural details of the error model. It is defined by a (probabilistic) machine over the error states declared in the error model type. Transitions between states can be triggered by error events, reset events, and error propagations.

Error events are internal to the component; they reflect changes of the error state caused by local faults and repair operations, and they can be annotated with occurrence distributions to model probabilistic error behavior.

Moreover, *reset* events can be sent from the nominal model to the error model of the same component, trying to repair a fault which has occurred. Whether such a reset operation is successful has to be modeled in the error implementation by defining (or omitting) corresponding state transitions.

Outgoing *error propagations* report an error state to other components. If their error states are affected, the other components will have a corresponding incoming propagation.

Figure 5.7 presents a basic error model for the battery device. It defines a probabilistic error event, `fault`, which occurs once every 1000 time units on average. Whenever this happens, the error model changes into the `dead` state. In the latter, the battery failure is signaled to the environment by means of the outgoing error propagation `batteryDied`. Moreover, the battery is enabled to receive a **`reset`** event from the nominal model to which the error behavior is attached. It causes a transition to the `resetting` state, from which the battery recovers with a probability of 20%, and returns to the `dead` state otherwise.

Just as for nominal component specifications, we distinguish between **`initial`** and **`activation`** starting states. Their meaning is similar to that of initial and activation modes: if an initial state is given, the error model is put in that state only in the beginning of system execution, supporting *error history* during deactivation phases. With an activation state, the error

```
error model BatteryFailure
  features
    ok: activation state;
    dead: error state;
    resetting: error state;
    batteryDied: out error propagation;
end BatteryFailure;

error model implementation
  BatteryFailure.Imp
  events
    fault: error event occurrence poisson 0.001;
    works: error event occurrence poisson 0.2;
    fails: error event occurrence poisson 0.8;
  transitions
    ok -[fault]-> dead;
    dead -[batteryDied]-> dead;
    dead -[reset]-> resetting;
    resetting -[works]-> ok;
    resetting -[fails]-> dead;
end BatteryFailure.Imp;
```

Figure 5.7: An Error Model.

model starts over again in that state after each (re-)activation of the respective component. This distinction is useful, e.g., for modeling the different error behavior of hardware and software components: while reactivating a hardware component (like a processor) will generally not remove the cause of an error, this is usually the case for software components (such as processes).

5.3 Fault Injection

As error models bear no relation with nominal models, an error model does not influence the nominal model unless they are linked through *fault injections*.

A fault injection describes the effect of the occurrence of an error on the nominal behavior of the system. More concretely, it specifies any of the following when the associated error models enters a specific error state:

- The value update that a data element of the component implementation undergoes;
- The list of nominal modes that are allowed to be entered;
- The list of events that can no longer occur.

To this aim, a nominal model can be associated with an error model, and subsequently the various fault injections can be specified. Multiple instances of these fault injections are possible

per nominal component, as long as each such instance does not overlap with an existing one. These fault injections are specified as follows:

- A fault effect consists of an error state s, an outgoing data port or subcomponent d, and the fault effect given by the expression a. Here d must not be a `clock` or `continuous` data subcomponent. While the error model is in state s, a is applied to d.
- A forced mode list consists of an error state s and a list of modes from the nominal components M. Only those modes in M can be active while the error model is in state s. When entering s and the current mode is not in M, the mode transitions to the first mode specified in M.
- A event inhibition is specified by an error state s and a list of inhibited events E, where each event is an outgoing event (data) port. While the error model is in state s, no transition triggered by an event in E can be taken.

Section 6.3 describes how to define fault injections using the graphical user interface of the COMPASS toolset.

The automatic procedure that integrates both models and the given fault injections, the so-called *model extension*, works as follows. The principal idea is that the nominal and error model are running concurrently. That is, the state space of the extended model consists of pairs of nominal modes and error states, and each transition in the extended model is due to a nominal mode transition, an error state transition, or a combination of both (in case of a reset operation).

For more specific information about the model extension semantics and possible fault injections, refer to [12], which describes these in fault detail, including their formal semantics.

Going back to the power system example above, the battery error model, `BatteryFailure.Imp`, can be associated with the battery device `Battery.Imp`. The association in and of itself will not change the behavior of the battery, but simply attach the error model. Fault injections are optional and specified separately.

To model a change in data values, some fault effects can be added. For example, in the error states `dead` and `resetting`, the value of `voltage` can be set to `0` by means of two fault effects (one for each error state).

In addition, it is possible to force the battery in the `depleted` state by means of forced modes: again for both the `dead` and `resetting` error states, the forced mode list could consist of solely the `depleted` state. Were the battery in the `charged` state, upon entering the error state `dead` or `resetting`, it would be forced into the `depleted` state.

Finally, modeling a more severe error condition, event inhibition allows the `empty` event to be inhibited, which will prevent it from being triggered while the battery is in the depleted state. Again in the same fashion for the other fault injections, specifying the `empty` event in the list of inhibited events for both the `dead` and `resetting` error states will prevent it from occurring. Note however, that in the given example implementation of the battery, a forced mode transition will bypass this event regardless.

Chapter 6

Handling Models

The *Model* pane of the toolset is displayed on startup, allowing the user to load files containing nominal and error model specifications and to relate these by defining fault injections. It is depicted in Figure 3.1.

6.1 Loading Models

Adding Models First, nominal and error model specifications have to be entered using a text editor, and saved to one or several files. The syntax of the corresponding SLIM Language and the meaning of its constructs is described in Document [12]. Files can then be loaded by clicking the *Add* button below the *Loaded Files* section of the window. This opens the *Open SLIM Files* window through which the file system is accessible. After selecting one or more files (using the *Shift* and/or *Control* key), they can be loaded by clicking *Open*. This procedure can be iterated until all required files have been added. The mode loading functionality is also accessible via the *Load SLIM model* entry of the *File* menu.

Syntactic Errors Loading fails if syntactic errors are detected. For example, a keyword might be misspelled, or a part of the specification tries to redefine an existing type or implementation of a component or error model that has already been loaded. In this case, diagnostic messages are displayed in the *Output Console*. After correcting the errors using the text editor, files can be reloaded using the *Reload all* button. The same is possible after modifications of the specification. Moreover, entries can be removed from the list of loaded files by marking them and clicking the *Remove* button.

Choosing the Root Component In the next step, the root component has to be chosen. This is necessary as the definition of fault injections (see below) and the subsequent analyses refer to the system that consists of the root component and all of its (direct and indirect) sub-components. The root component candidates as displayed in the *Root* section are those component implementations that do not occur as a sub-component of any other component, and that are not contained in any package. If this candidate is unique, it is selected by default.

FDIR Components Components that have been designated as part of the FDIR specification and optionally be replaced by an empty implementation. This can be used for instance to verify the FDIR component does not alter the nominal behavior. Using the checkboxes next to

the FDIR components listed will toggle between the original and empty implementation. The empty implementation corresponds to that of a subcomponent which refer to a type, see [12] for details on the semantics.

6.2 Saving Models

Accessible from the *CSSP* tab, the *Save* button allows the user to save the current loaded model, its properties and fault injections. When saving a model, a dialog will pop-up asking for the name of the file to save the model to. By default, this is a unique name, but can be changed as necessary. Note however that saving a model over an existing file will overwrite it.

Models are always saved into a single file. This means that when multiple model files were loaded before, the contents of these files all get saved to the new model file. Furthermore, any custom formatting or comments are lost. Therefore, it is recommended to save to a new file and adjust its contents with an editor as necessary.

6.3 Defining Fault Injections

After successfully loading nominal and error model specifications, both can be related by adding fault injections. The purpose of the latter is twofold:

- They associate an error model with an instance of a nominal component implementation.
- They define the impact of the occurrence of an error to the nominal behavior of the respective component.

Entering Fault Injections The *Fault Injection* dialog (as depicted in Figure 6.1) is accessible via the *Edit* button in the *Fault Injections* section of the main window.

On the left, the component to which the fault injection should apply can be selected. This can be a component type, implementation or subcomponent.

On the right, the associated error model and fault injections themselves can be specified. The error model can be selected on the top, and should be specified before any fault injection. Setting it to `None` disables fault injection for that component.

All fault injections have in common they specify an error state, which is one of the states specified for the chosen error model. As starting error states are considered to be nominal, they are not shown in the list.

The *Fault Effects* section provides the list of specified fault effects (which initially is empty), to which effects can be added with the *Add* button. They consist of:

- An error state;
- A target: This input allows to select the component's data element (data subcomponent or outgoing data port) that is affected by the error that was chosen from the *State* list. Fault injections cannot be defined for `clock` or `continuous` data subcomponents.
- An effect: In this input one has to enter the right-hand side of the assignment that defines the failure effect by overriding the nominal behavior. It must be a well-typed expression over the incoming data ports of the nominal component if the fault injection affects a flow, and a well-typed expression over arbitrary data elements otherwise.

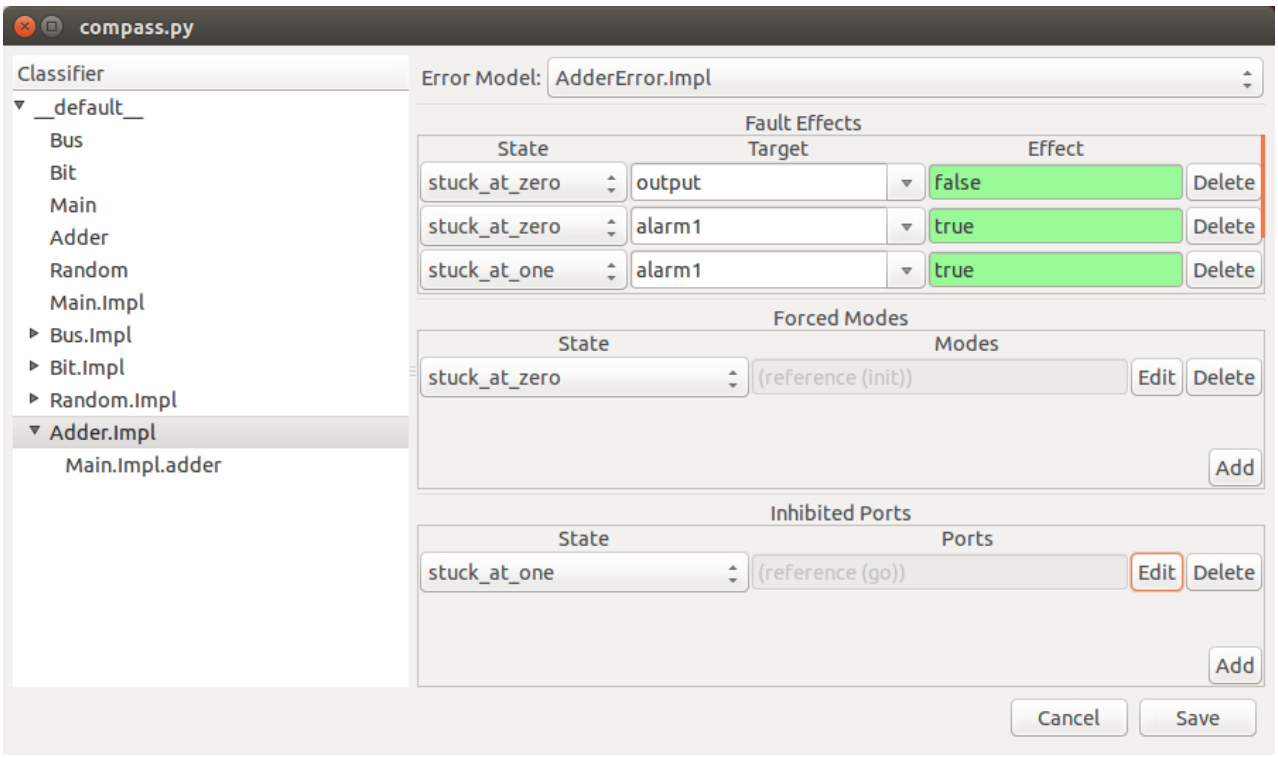


Figure 6.1: The *Fault Injection* dialog

The *Forced Modes* section provides the list of specified forced modes (which initially is empty), to which new entries can be added with the *Add* button. They consist of:

- An error state;
- A list of modes which are permitted in the selected state.

The *Inhibited Ports* section provides the list of specified inhibited ports (which initially is empty), to which new entries can be added with the *Add* button. They consist of:

- An error state;
- A list of outgoing event (data) ports which are inhibited (cannot occur) in the selected state.

After providing all required information, the fault injections can be saved by clicking *Save*. Moreover changes to the fault injections can be reverted at any time by clicking *Cancel*. Figure 6.2 shows the toolset after loading a model and defining some fault injections.

It is possible to add several fault injections. However,

- at most one error model can be attached to each component, and
- at most one injection per error state and data element can be defined.

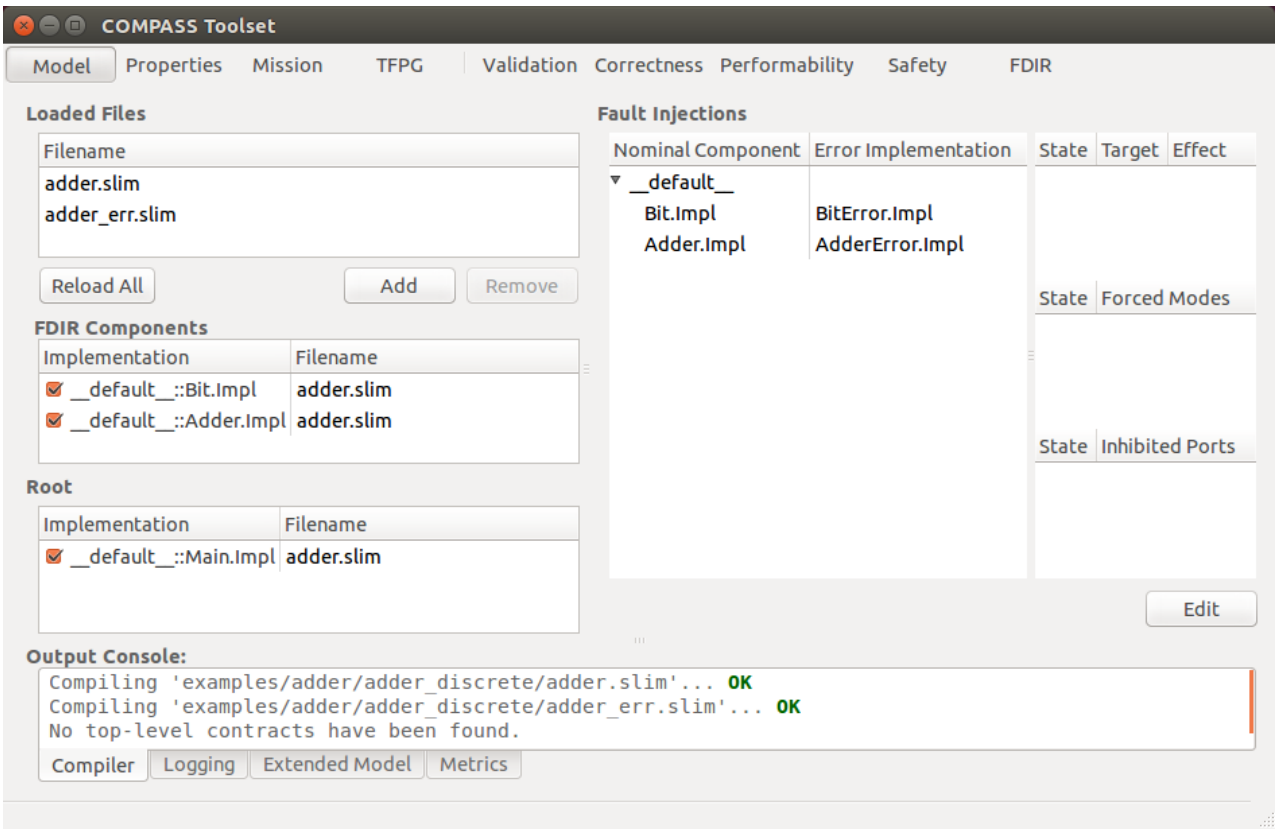


Figure 6.2: Main Window of COMPASS Toolset After Model Loading

Model Extension After providing both nominal and error model specifications and fault injections, these parts are automatically combined in the toolset using a procedure that is called *model extension* (cf. Section 5.3). The resulting extended model is then used in the subsequent analysis phases. It is not mandatory, however, to apply all fault injections that have been defined so far in the model extension step: a subset can be chosen by (un)checking the corresponding boxes in the *Use* column of the *Fault Injections* section.

Saving and Loading Fault Injections Additional functionality is provided through the *File* menu, which provides two entries for respectively saving and loading fault injections. When using the first, only those fault injections that are checked in the *Use* column are stored. The default file extension is "`.fixml`" (for "Fault Injection XML file").

Chapter 7

Properties

Various analysis function of the COMPASS toolset require, aside from the system specification, one or more properties. This section describes the possible properties that can be specified.

7.1 Atomic Propositions

A core concept of properties is the atomic proposition. An atomic proposition is a Boolean expression, similar to a mode transition guard. They define a certain state in the model, generally by comparing a variable.

Example atomic propositions are:

- `temp > 11`, where `temp` is an outgoing integer data port of the root component,
- `mode = mode:Ok`, where `Ok` is a possible mode of the root component,
- `sensor.mode = mode:Failed`, where `Failed` is a possible mode of the `sensor` component,
- `error = error:transmittedFault`, where `transmittedFault` is a possible state of the error model that is associated with the root component, and
- `status = enum:ACK`, where `status` is a data subcomponent of the root component of an enumeration type with possible value `ACK`.

We call all of the atomic propositions above *instance based*. This is because they talk about e.g. a data subcomponent of a certain component implementation *instance*. In order to talk about this instance one has to provide the unique path from the root component to that instance. For example, in `sc.ssc.port1`, `sc` is a subcomponent instance of the root and `ssc` is a subcomponent instance of `sc`.

Operators The following operators may be used, ordered from higher precedence to lower:

1. `not` (negation), `-` (unary minus) and `(...)`,
2. `*` (multiplication), `/` (division), `mod` (modulo),
3. `+` (addition), `-` (substraction),

4. `>` (greater than), `<` (less than), `>=` (greater or equal than), `<=` (less or equal than), `=` (equal to), `!=` (not equal to),
5. `and`
6. `or`, `xor` (exclusive or), `xnor` (exclusive not or)
7. `iff` (if and only if),
8. `imp` (implies),
9. `case` (conditional),

Note that the overall type of the atomic proposition must be Boolean. Please see the SLIM language specification [12] for a complete overview of available data types and operators.

Identifiers and Values Atomic propositions may refer to the following objects:

- Ports, e.g. `sc.ssc.port1`, where `sc` is a subcomponent identifier, `ssc` is a subcomponent of the implementation of `sc` and `port1` is the port identifier.
- Data subcomponents, e.g. `sc.data1`, where `sc` is a subcomponent identifier and `data1` is the data subcomponent identifier.
- Constant values:
  - Integers, e.g. `42`,
  - Reals, e.g. `42.001`,
  - Booleans, e.g. `true`,
  - Enumeration literals, e.g. `enum:C1`, where `C1` is the literal.
- Mode variables, e.g. `sc.mode` where `sc` is a the identifier of a subcomponent of the root component, representing the current mode of the subcomponent `sc`.
- Mode names, e.g. `mode:m1`, where `m1` is the mode.
- Error variables, e.g. `error`, referring to the current state of the error model that has been associated with the root component.
- Error state names, e.g. `error:e1`, where `e1` is the error state.

7.2 CSSP

Properties can be specified by means of the CSSP. the CSSP defines a set of model attributes, which can be used to derive a formal property. Using the CSSP simply requires setting such attributes, the property will then automatically be determined.

The following table lists all the available properties. These properties can be specified in the model directly, or set from, the GUI (cf. section 7.5). The actual formal definitions of these properties and their meaning follow after the table.

Name	Value type	Applies to
Change	reference(event port, event data port)	data, data port
ModeInhibited	list of reference (event port, event data port)	mode
ModeInvariant	aadlstring	mode
MonitorRange	range of aadlinteger	event data port
MonitorResponse	reference(event port, event data port)	event data port
MonitorDelay	Time	event data port
MonitorEnabled	list of reference(mode)	event data port
AlarmDelay	Time	event port, event data port
RecoveryDelay	Time	event port, event data port
Timeout	Time	event port, event data port
TimeoutReset	reference(event port, event data port)	event port, event data port
TimeoutCondition	list of reference(mode)	event port, event data port
Function	aadlstring	data port, event data port
InvariantRange	range of aadlinteger	data port, event data port
Reaction	reference(event port, event data port)	event port, event data port
ReactionCondition	list of reference(mode)	event port, event data port
ReactionMinDelay	Time	event port, event data port
ReactionMaxDelay	Time	event port, event data port
PrecededBy	reference(event port, event data port)	event port, event data port
PrecededCondition	list of reference(mode)	event port, event data port
PrecededMinDelay	Time	event port, event data port
PrecededMaxDelay	Time	event port, event data port
PeriodInterval	Time	event port, event data port
PeriodOffset	Time	event port, event data port
PeriodJitter	Time	event port, event data port
PeriodEnabled	list of reference(mode)	event port, event data port
ThroughputInput	reference(event port, event data port)	event port, event data port
ThroughputRatio	aadlinteger	event port, event data port
Tolerance	aadlinteger	port
FailureCondition	list of reference(mode)	component

The following table provides an overview of the formal properties defined by the CSSP. In the left column, the name of the formal property is given, which can be referenced from contracts or generic properties (cf. section 7.4). On the right, the formal definition is given in terms of the CSSP properties.

Name	**element**	**Formal property**
PersistentProperty(p)	data	$\mathrm{G}(\mathit{change}(p) \rightarrow \mathbf{Change}(p))$
Specifies that the value of p changes only on the $\mathbf{Change}(p)$ event.		
ModeInhibitedProperty(m)	mode	$\mathrm{G}(\mathsf{mode} = m \rightarrow \bigwedge_{e \in \mathbf{ModeInhibited}(m)} !e)$
Specifies that the events in $\mathbf{ModeInhibited}(m)$ cannot occur in mode m.		
ModeInvariant-Property(m)	mode	$\mathrm{G}(\mathsf{mode} = m \rightarrow \mathbf{ModeInvariant}(m))$
Specifies a generic invariant for mode m.		

MonitorProperty(p)	event data	$\mathrm{G}((p \wedge \mathsf{mode} \in \mathbf{MonitorEnabled}(p) \wedge (\mathit{data}(p) \notin \mathbf{MonitorRange}(p))) \rightarrow \mathrm{F}_{\leq u}\, \mathbf{MonitorResponse}(p))$, where $u = \mathbf{MonitorDelay}(p)$.
Specifies that the event **MonitorResponse**(p) is fired if the value of p falls outside the specified **MonitorRange**(p).		
CompleteAlarmProperty(p)	event (data)	$\mathrm{G}(\mathit{rise}(\mathsf{mode} \in \mathbf{FailureCondition}(p)) \rightarrow \mathrm{F}_{\leq u}\, p)$, where $u = \mathbf{AlarmDelay}(p)$.
Specifies that if failure configuration **FailureCondition**(p) is entered, the alarm event p follows.		
CorrectAlarmProperty(p)	event (data)	$\mathrm{G}(p \rightarrow \mathrm{O}_{\leq u}\, \mathit{rise}(\mathsf{mode} \in \mathbf{FailureCondition}(p)))$, where $u = \mathbf{AlarmDelay}(p)$.
Specifies that if the alarm event p occurs, it was preceded by entering the failure configuration **FailureCondition**(p).		
RecoveryProperty(p)	event (data)	$\mathrm{G}(p \rightarrow \mathrm{F}_{\leq u}\, \mathsf{mode} \notin \mathbf{FailureCondition}(p))$, where $u = \mathbf{RecoveryDelay}(p)$.
Specifies that upon event p, eventually the failure configuration **FailureCondition**(p) is recovered.		
CompleteTimeoutProperty(p)	event (data)	$\mathrm{G}(F_{\leq u}(\mathbf{TimeoutCondition}(p) \rightarrow p \vee \mathbf{TimeoutReset}(p)))$, where $u = \mathbf{Timeout}(p)$.
Specifies that if p does not occur within **Timeout**(p), the alarm **TimeoutReset**(p) must occur		
CorrectTimeoutProperty(p)	event (data)	$\mathrm{G}(\mathbf{TimeoutCondition}(p) \wedge \mathrm{O}_{\leq u}\, \mathbf{TimeoutReset}(p) \rightarrow !p)$, where $u = \mathbf{Timeout}(p)$.
Specifies that if the alarm **TimeoutReset**(p) occurs, the event p did not occur.		
FunctionProperty(p)	data	$\mathrm{G}(p = \mathbf{Function}(p))$
	event data	$\mathrm{G}(p \rightarrow \mathit{data}(p) = \mathbf{Function}(p))$
Specifies the value of p remains within the associated function **Function**(p) (an expression).		
InvariantProperty(p)	data	$\mathrm{G}(p \in \mathbf{InvariantRange}(p))$
	event data	$\mathrm{G}(p \rightarrow \mathit{data}(p) \in \mathbf{InvariantRange}(p))$
Specifies the value of p remains within the associated range of values.		
ReactionProperty(p)	event (data)	$\mathrm{G}((p \wedge \mathsf{mode} \in \mathbf{ReactionCondition}(p)) \rightarrow \mathrm{F}_{\in I}\, \mathbf{Reaction}(p))$, where $I = [\mathbf{ReactionMinDelay}(p), \mathbf{ReactionMaxDelay}(p)]$.
Specifies the event p is followed by **Reaction**(p) provided the mode is in **ReactionCondition**(p).		
PrecededByProperty(p)	event (data)	$p \rightarrow \mathrm{O}_{\in I}\, \mathbf{PrecededBy}(p))$, where $I = [\mathbf{PrecededMinDelay}(p), \mathbf{PrecededMaxDelay}(p)]$.
Specifies the event p is preceded by **PrecededBy**(p)		
PeriodProperty(p)	event (data)	$(\mathrm{F}_{\leq v}!\mathit{enabled} \vee \triangleright_{[v,v+j]}\, p) \wedge \mathrm{G}(\mathit{rise}(\mathit{enabled}) \rightarrow (\mathrm{F}_{\leq v}!\mathit{enabled} \vee \triangleright_{[v,v+j]}\, p)) \wedge \mathrm{G}((p \wedge \mathit{enabled}) \rightarrow (\mathrm{F}_{\leq u}!\mathit{enabled} \vee \not\triangleright_{[u,u+j]}\, p))$

		where u = **PeriodInterval**(p), v = **PeriodOffset**, j = **PeriodJitter**, *enabled* = mode $\in$ **PeriodEnabled**.
Specifies the event p occurs within the specified period and optional offset		
ThroughputRatio(p)	event (data)	**PeriodInterval**(p) = **ThroughputRatio**(p) $*$ **PeriodInterval**(**ThroughputInput**(p))
Specifies the throughput of event p as an ratio of the throughput of **ThroughputInput**(p).		
ToleranceProperty(p)	port	G($\sharp p <=$ **Tolerance**(p))
Specifies the tolerated number of failure events **Tolerance**(p). This is generally an assumption for other properties.		
MTTF(x)	event, component	ExpectedTime(mode $\in$ **FailureCondition**(x))
The expected time (mean time) until **FailureCondition**(x) holds.		
MTTR(x)	event, component	ExpectedTime(mode $\notin$ **FailureCondition**(x)) with starting state = **FailureCondition**(x)
The expected time (mean time) until **FailureCondition**(x) no longer holds.		
Availability(s)	system	LRA(mode $\notin$ **FailureCondition**(s))
The availability specified as the *long-run average* of s being in a nominal mode.		

For more details about the CSSP, please refer to [8].

7.3 Property Patterns

A property pattern uses a predefined property structure with a number of placeholders, which may be filled in by the user. A formal definition of the property is then automatically derived.

Often, placeholders are defined as *atomic propositions*, which are Boolean SLIM expressions that define a certain state in the model, generally by comparing a variable. The following are some examples of such atomic propositions (see [12] for more details):

- `true` : Simply true
- `output = 3` : Asserts the output variable is 3
- `subcomponent.error = error:failed` : Asserts the error state of some subcomponent is the state labeled `failed`

In general, a pattern is structured as follows:

- An (optional) scope, which defines when the property should hold. The scope can have a beginning and end, which are specified by means of atomic propositions.
- The class of the patterns. These classes are described in the following.
- An optional time bound of the pattern. This time bound specifies during what time the property must be true.

- An optional probability of the pattern. The probability specifies the likelihood the property must be true. Currently its value is ignored: performability analysis simply calculates the probability itself.

7.3.1 Pattern classes

Patterns are primarily defined by their class, which can be further grouped into two subclasses: *Occurrence* and *Order*. The Occurrence group of classes consists of the following patterns:

- Universality: States a property always holds true;
- Absence: States a property never holds true;
- Existence: States a property eventually holds true;
- Recurrence: States a property holds true recurringly, that is, each time it becomes false, it will become true again later on.

The Order group of classes consists of these patterns:

- Precedence: Some property must always be preceded by another.
- Response: Some property must always be followed by another.
- Response invariance: Some property must always be followed by another, which from that point onwards always holds true.
- Until: Some property must always be followed by another, and must hold true until that happens.

7.4 Generic Properties

Generic properties are specified directly as (temporal) SLIM expressions. They can be useful for the following purposes:

- Specify a propositional property;
- Specify expected time or long-run average properties: Using the `ET` and `LRA` operators, these properties can be formulated. For example, `ET error = error:Dead` or `LRA error = error:normal`.
- In case the CSSP or Pattern based approaches prove to be too restrictive.

7.4.1 Propositional Properties

For particular analyses, like fault tree analysis or failure modes and effects analysis, one has to express purely propositional properties. Such a property corresponds to a valid atomic proposition, like `error = error:transmittedFault`.

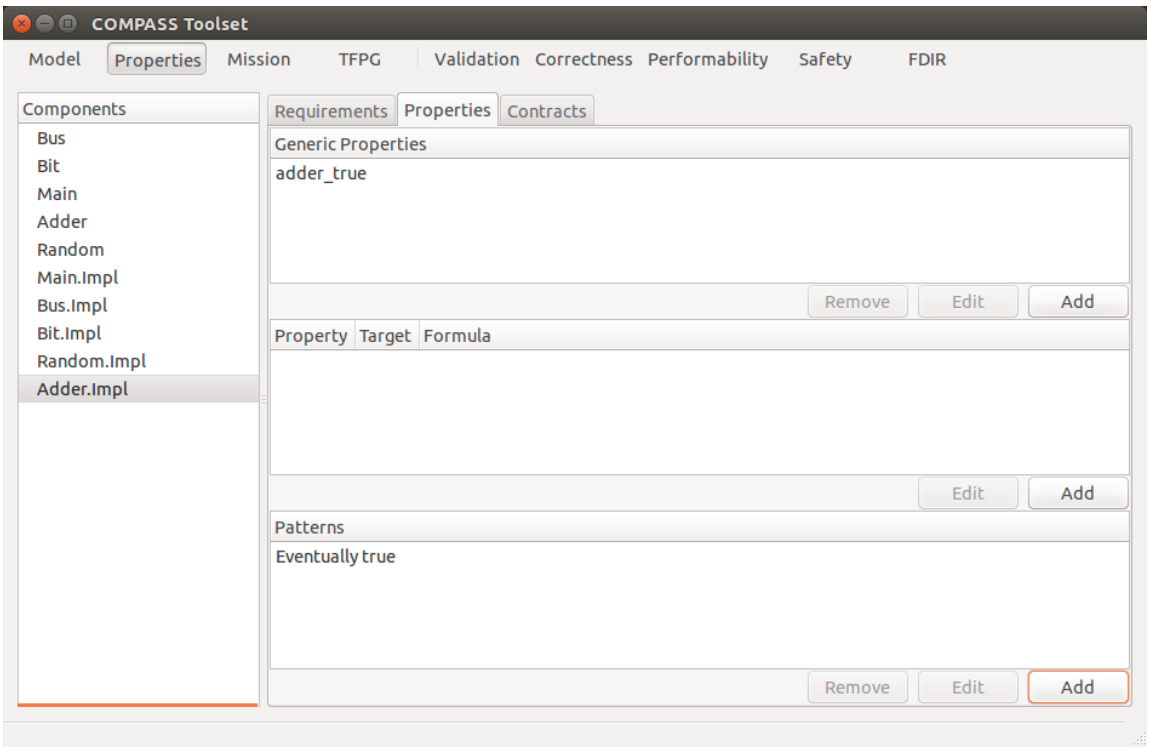


Figure 7.1: Property Manager View

7.5 GUI-Based Property Management

The *Properties* view of the GUI provides an overview of the specified properties, as well as the means to add, edit or delete them. See Figure 7.1 for an example view.

The property manager consists of primarily a treeview on the left, listing the components of the system specification, and the property specification panels on the right. Three panels are available:

- Requirements: Here, model requirements can be specified(which are free-form text) which may reference a property. This is for traceability of the model. Furthermore, a property specification wizard is available from this panel, which provides a simple step-by-step approach for specifying properties based on requirements.
- Properties: Here, the actual (formal) properties can be defined, in the three aforementioned ways:
  - Generic Properties, see Figure 7.2;
  - CSSP Properties, see Figure 7.3;
  - Pattern Properties, see Figure 7.4.
- Contracts: Here, contracts and contract refinements can be specified.

Entering Pattern Properties Here we present an example of how to create and edit properties using the property manager. Before doing this, we have to load a SLIM model.

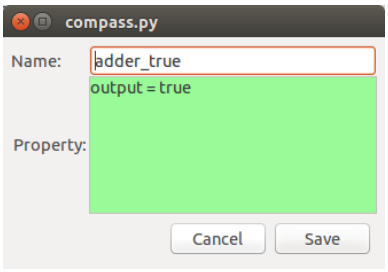


Figure 7.2: Generic Property specification dialog

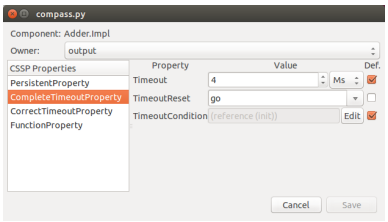


Figure 7.3: CSSP specification dialog

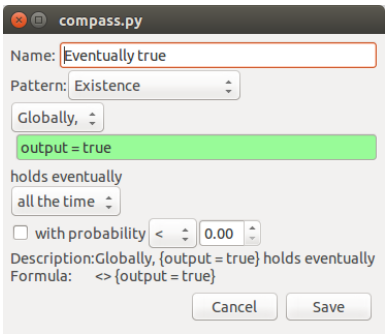


Figure 7.4: Pattern Property specification dialog

If we want to talk about error states in our properties, we additionally have to specify fault injections. We will present a stepwise example using a sensor-filter example:

1. Load `sensorfilter.slim` and `sensorfilterErr.slim` from the `documentation/examples/sensorfilter` folder in the COMPASS toolset.
2. Create a fault injection with implementation `SensorFailures.Impl`, state `Drifted1`, Component `Sensor.Impl`, data element `output` and effect `output *2`.

Now the sensorfilter model is loaded and fault injections over it are specified. First we define some (arbitrary) requirements for the model, for which we specify some properties later on. For this purpose, the following three requirements are defined:

1. *Requirement 1*: The probability of a sensor drifting within 50 time units must be lower than 50%.
2. *Requirement 2*: The output of a sensor under nominal conditions must be within $[0, 5]$.
3. *Requirement 3*: The expected time until a sensor fails must be larger than 10 time units.

The next step is to create properties for each requirement. For the first requirement, we make use a of a pattern property:

1. Switch to the *Properties* view.
2. In the left treeview, select `Sensor.Impl`
3. Select the *Properties* subview.
4. In the *Pattern* panel, click *Add*.
5. Enter at the top a name for the property, for example `Eventually drifted`.
6. Choose `Existence` and `Globally` form the first two drop-down lists.
7. Fill in the gap (atomic proposition) of the pattern, for example `error = error:Drifted1`.
8. Change `all the time` into `before`, and enter `50`. This sets the time bound.
9. Check the box next to `with probability`. The operator and value do not matter.
10. Click the *Save* button which becomes sensitive if all atomic propositions are syntactically correct. In case there are errors, a tooltip will automatically appear and display an error message in case it is possible to provide one.

The property is now added to the model. This should be followed by adding the requirement:

1. Select to the *Requirement* subview.
2. Click *Add*.
3. In the *Requirement* field, enter "Requirement 1: The probability of a sensor drifting within 50 time units must be lower than 50%."
4. In the *Formalization* field, enter `Eventually drifted`.
5. Click *Save*.

For the second requirement, we make use of the CSSP:

CSSP Properties For the CSSP, the procedure is slightly different. The CSSP is specified by first selecting a element of the component for which to specify the CSSP property (or the component itself, which is the default). Then a CSSP property is selected, followed by assigning values to its attributes.

CSSP attributes have a checkbox *Def.* next to them. This is to indicate the attribute is set (or not). To unset an attribute, simply ensure the checkbox is not active.

1. Switch to the *Properties* view.
2. In the left treeview, select `Sensor.Impl`
3. Select the *Properties* subview.
4. In the *CSSP* panel, click *Add*.
5. Set the *Owner* to `Cycle`.
6. Select *ModeInvariantProperty* in the left list.
7. Set the *ModeInvariant* field to `error = error:OK implies output <= 5`
8. Ensure the *Def.* box is checked.
9. Click the *Save* button.

The CSSP property is now defined. This should be followed by adding the requirement:

1. Select to the *Requirement* subview.
2. Click *Add*.
3. In the *Requirement* field, enter "Requirement 2: The output of a sensor under nominal conditions must be within [0, 5]".
4. In the *Formalization* field, enter `ModeInvariantProperty`.
5. Click *Save*.

Generic Properties For the third and final requirement, we enter a property directly, in this case an expected time property:

1. Switch to the *Properties* view.
2. In the left treeview, select `Sensor.Impl`
3. Select the *Properties* subview.
4. In the *Generic* panel, click *Add*.
5. Enter at the top a name for the property, for example `Expected time failed`.
6. Enter `ET(error != error:OK)` in the *Property* field.

7. Click the *Save* button which becomes sensitive if all atomic propositions are syntactically correct.

The property is now added to the model. This should be followed by adding the requirement:

1. Select to the *Requirement* subview.
2. Click *Add*.
3. In the *Requirement* field, enter "Requirement 3: The expected time until a sensor fails must be larger than 10 time units."
4. In the *Formalization* field, enter `Expected time failed`.
5. Click *Save*.

Editing Properties can be edited. Simply double click on it, or select it and click *Edit*. Note that whenever a property is edited all analysis results attached to it are lost.

Deleting To delete a property, first click on the property to be deleted and then click the *Delete* button. Note that for the CSSP, all of its associated attributes are unset, meaning other CSSP properties may become deleted as well.

Loading/Saving Properties are loaded and saved together with the model. Therefore, to save the properties, simply select *Save SLIM Model As...* from the *File* menu. Note that any analysis results are not saved along with the property.

Older version of COMPASS used an XML based format for storing properties. To load these files, select the item *Load Properties...* from the *File* menu. These properties are then attached to the current root component of the model.

Chapter 8

Mission Specification

The *Mission* is the tab that contains modeling specifications that concern mission characteristics such as mission phases, operational modes and spacecraft configurations.

Remark Mission specification is currently not used in other analyses of the COMPASS toolset. It is present mainly for future extensions. For instance, it is conceivable to use of mission phases and operational modes for verification activities such as, for instance, FTA and definition of FDIR specifications. In this way, it would be possible to generate artifacts (e.g., FTs) that refer to given mission phases or operational modes.

COMPASS provides:

Mission Phases Mission phases or Phases are represented in a symbolic way. Phases can be defined in terms of symbolic expressions over system/components configurations, that are modeled in SLIM.

Operational Modes Operational modes or Op-modes are represented in a symbolic way. Operational modes can be defined in terms of symbolic expressions over system/components configurations, that are modeled in SLIM.

Spacecraft Configurations Spacecraft configurations, in combination with mission phases and operational modes, can be used as targets of reconfiguration operations.

Steps We are now describe the mission definition in more detail. In particular, we describe the following steps:

- Define list of Phases and Op-modes names
- Define S/C Configurations and associate them to Op-modes
- Define Phase/Op-mode combination via Observable

8.1 Loading and Saving the Mission Specification

Mission files are defined in XML format (extension: `.mxml`); once they are created, they can be loaded into the toolset.

Steps

- Select the SLIM model from the *Model* pane
- Load mission specification using the *File menu*
- The mission specification is now loaded in the tab (see Figure 8.1)

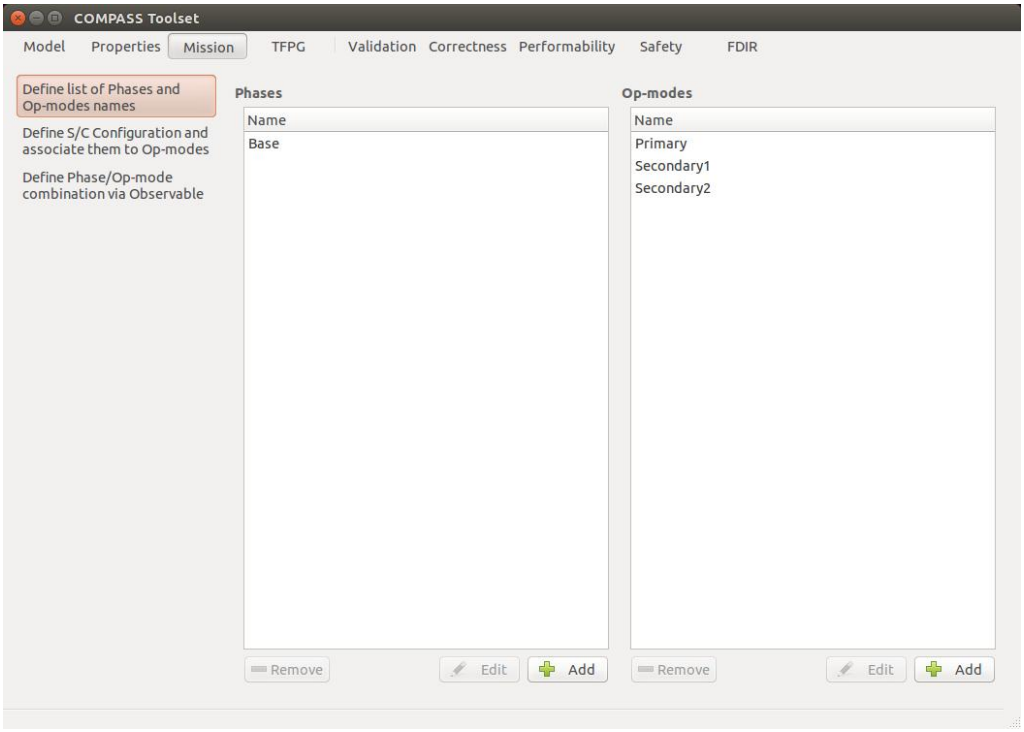


Figure 8.1: The *Mission* tab

Mission specifications can now be edited as shown in Figure 8.2 (in this case, the Mission configuration is beeing edited), removed and then saved to file.

8.1.1 Phases and Op-modes names

Phases and operational modes are defined as a simple list of names (see Figure 8.3). There is no semantic behind these names, and they are not related to the SLIM Model.

The COMPASS Toolset allows the user to enter new phases and operational modes, to edit some previously loaded specifications (i.e. change the name) and to remove them.

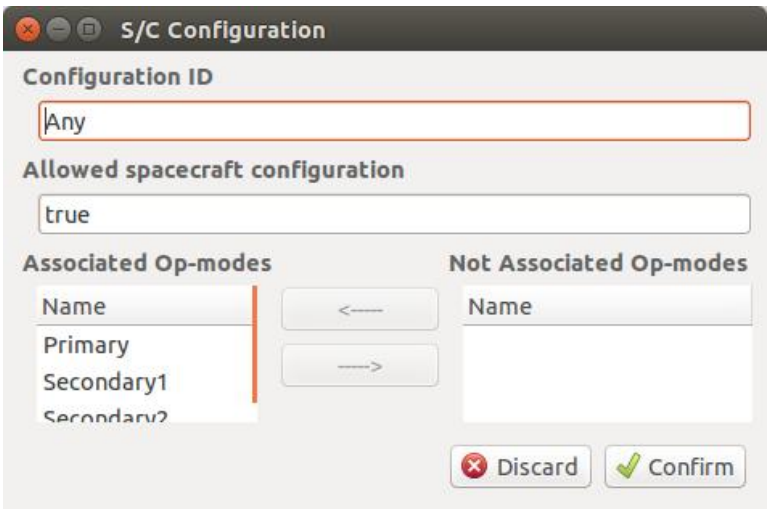


Figure 8.2: Editing *Mission* configuration

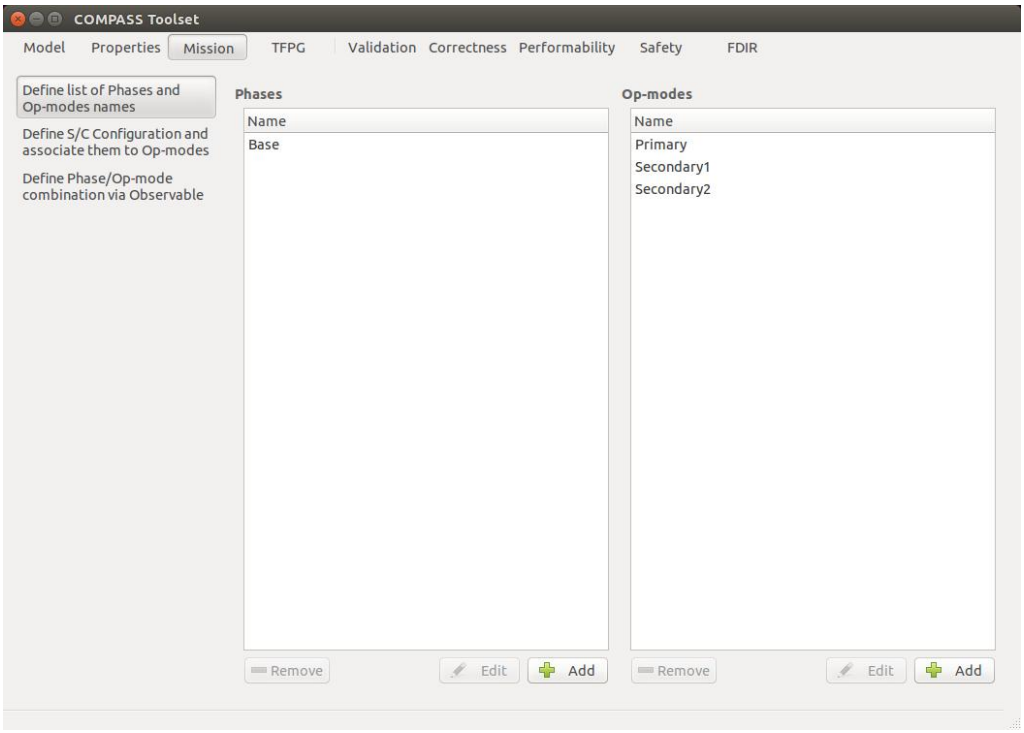


Figure 8.3: Define List of Phases/Op-modes names

8.1.2 S/C Configurations associated to Op-modes

Each S/C Configuration is represented by a label and a formula expressed over the SLIM model (see Figure 8.4 for an example). No particular requirement is given on these configurations; in particular, we do not care if configurations are overlapping. It is performed a syntactic check on these definitions, in order to ensure that the specified components exist and checks that each configuration is associate to at least one op-mode.

The COMPASS Toolset allows the user to enter new configurations, to edit and to remove them.

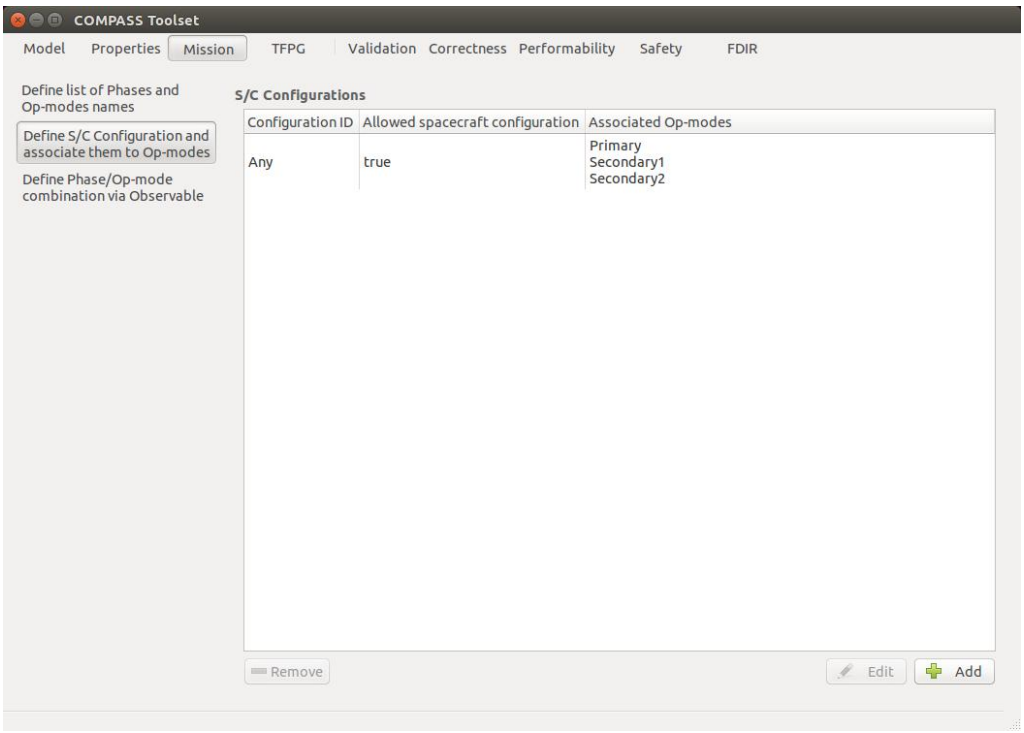


Figure 8.4: Define S/C Configurations and associate them to Op-modes

8.1.3 Phase/Op-mode Combination via Observable

The third sub-panel of the *Mission* tab allows to define the Phases in terms of Op-Modes and associate an expression over the observables to define a particular phase/op-mode combination (see Figure 8.5).

It is important that each pair of expressions identifies a disjoint set of states, therefore a check is performed prior to synthesis. All elements included in the definitions are tagged as observables and an error is raised if cells are overlapping.

In this case, the toolset allows the user to edit a previously loaded definition and association, but not to remove them or to create new ones.

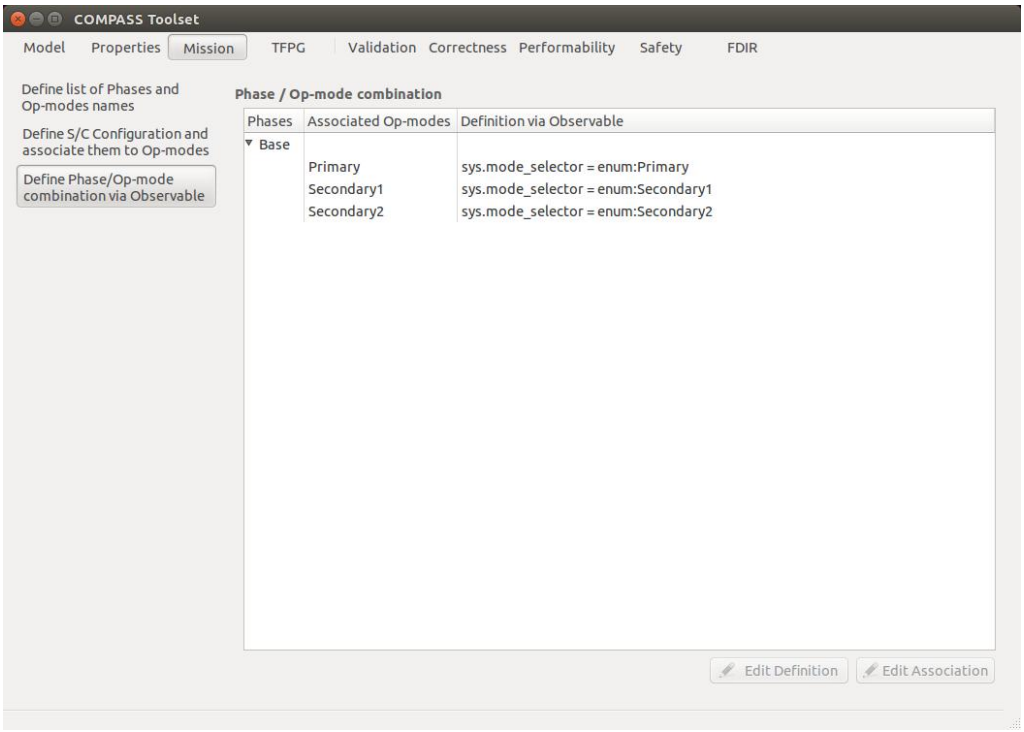


Figure 8.5: Define Phase/Op-mode Combination via Observable

Chapter 9

Analyses

9.1 Support of Aspects w.r.t. Analyses

The COMPASS toolset pushes supportable analyses over SLIM models to the current theoretical limits. For this reason, particular analyses can only handle a subset of SLIM constructs. Table 9.1 shows the list of analyses and indicates which part of the SLIM specification is covered. The toolset provides warnings and disables functionalities when such a limit is reached.

Discrete means that the SLIM model under verification only uses discrete data elements, which are integers, integer ranges and Boolean values. Hybrid means that the SLIM model under verification uses the full scope of the SLIM language, including clocks and continuous data elements.

Verifications that are marked with a "[1]" in Table 9.1 indicate that while the respective analysis is supported, its semantic interpretation in combination with hybrid aspects is still under debate.

9.2 Validation

The *Validation* window gives the user the possibility to validate the contracts that are specified in the input model. Three types of validations are possible:

1. Contract validation: Analyze the consistency, possibility of a scenario or if a set of properties entails an assertion.
2. Contract refinement: Perform analysis on contract refinements.
3. Contract tightening: Tightening of contract refinements.

9.2.1 Contract Validation

The contract validation panel (see Figure 9.1) can perform three different types on validation for a given component. The component to perform validation on can be selected from the left hand side.

Validation is performed on *contract properties*, which are the individual assumptions and guarantees of the contracts. These properties are listed as ASSUMPTION, GUARANTEE

Analysis	**Discrete**	**Hybrid**
TFPG Synthesis	☑	☑
TFPG Behavioral Validation	☑	☑
TFPG Effectiveness Validation	☑	☑
Contract Consistency Validation	☑	☑
Contract Possibility Validation	☑	☑
Contract Assertion Validation	☑	☑
Contract Refinement	☑	☑
Tightening of a Contract	☑	☑
Model Simulation	☑	☑
Deadlock Checking	☑	
Model Checking	☑	☑
Zeno Analysis	☑	☑
Time Divergence	☑	☑
Contract-Based Verification	☑	☑
Performability	☑	
Fault Tree Generation	☑	☑
Dynamic Fault Tree Generation	☑	
Hierarchical Fault Tree Generation	☑	☑
Failure Modes and Effects Analysis	☑	☑
Dynamic Failure Modes and Effects Analysis	☑	
(Dynamic) Fault Tree Evaluation	☑	☑[1]
Criticality Evaluation	☑	☑[1]
(Dynamic) Fault Tree Verification	☑	☑[1]
Fault Detection Analysis	☑	☑
Fault Isolation Analysis	☑	☑
Fault Recovery Analysis	☑	☑
Diagnosability Analysis	☑	☑
Fault Coverage Analysis	☑	☑

Table 9.1: Overview of analyses and supported modelling aspects.

and NORM_GUARANTEE (the norm guarantee, which is defined as ASSUMPTION → GUARANTEE) in the *Contract validation* panel.

First, contract validation checks whether individual contract properties are consistent, by selecting the Consistency validation type. This will check that contract properties are mutually consistent. If so, a witness trace will be given. If not, and enabled, an unsat core is generated instead. To check all contract properties of all components in the model, select *Consistency of all Contracts of all Components*.

The Possibility type validates if contract properties are consistent with a scenario, which is specified by means of another contract property. The inputs are the same as for the Consistency check, with the addition of a secondary property selected in the lower half of the panel. Similar to the Consistency validation, a witness is provided if the scenario is possible, and an unsat core if not.

The Assertion validation type checks if a set of properties entail an assertion, specified with a new property. The inputs are the same as for the Possibility validation type.

For all three types of analysis, the following options can be set:

- UnsatCores: If this option is check, unsat cores are generated when either an assert check succeeds, or a consistency or possibility check finds a counterexample.
- Consistency of all Contracts of All Components: If this options is checked, the entire architecture of the input model is validated. If unchecked, only the selected component is validated.
- Algorithm: The validation algorithm to use. For details, see the description of these options in Section 9.4.

The Properties list allows the selection of properties to be validated. For each contract of the selected component, the ASSUMPTION, GUARANTEE and NORM_GUARANTEE properties are available. In case all contract properties should be validated, the checkbox can be selected.

Steps

1. Select the SLIM model from the *Model* pane
2. In the *Validation* pane select the type (*Consistency*, *Possibility* or *Assertion*
3. Activate the checkbox *Consistency of all Contracts of all Components* (optional)
4. If the *Consistency* type has been selected, this will enable the button *Run*
5. If *Possibility* or *Assertion* type has been checked, select one or more contracts from the *Contracts* pane, and a list of properties in the proper panes; this will enable the button *Run*
6. The user can now run the analysis by clicking on the button *Run*, that will fill the pane below with the results of the computation

In this example (see Figure 9.2) the consistency type has been selected; a message confirms that all the contracts are consistent and the list of consistency component results is displayed. It is possible to see the different traces by clicking the buttons *Next* and *Prev*.

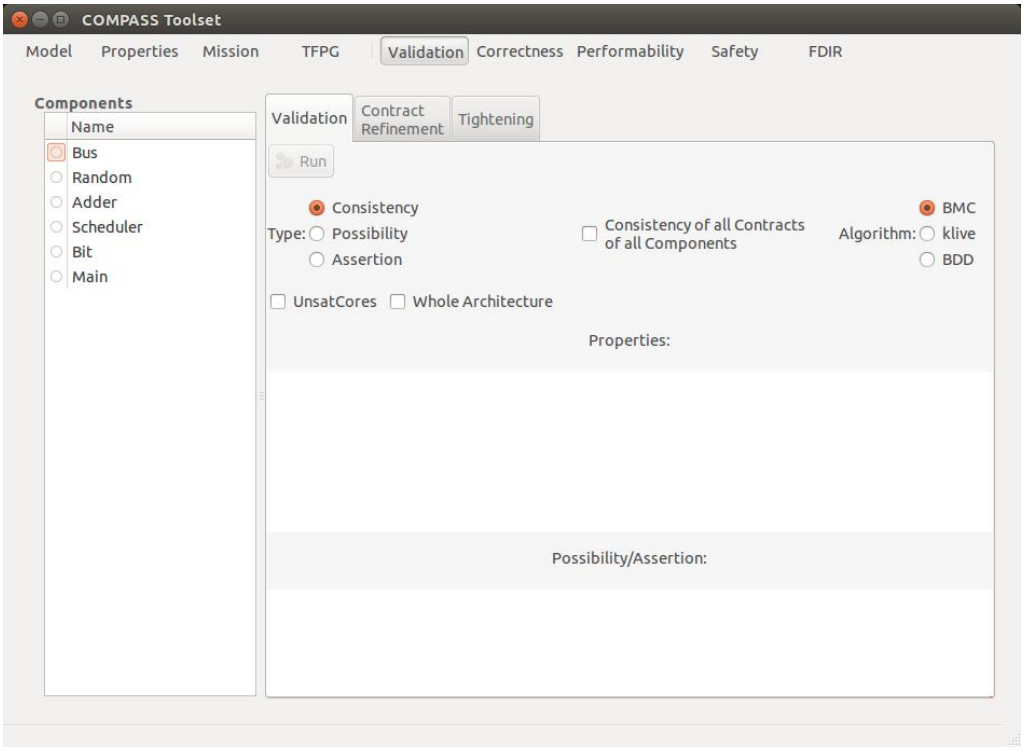


Figure 9.1: Validation

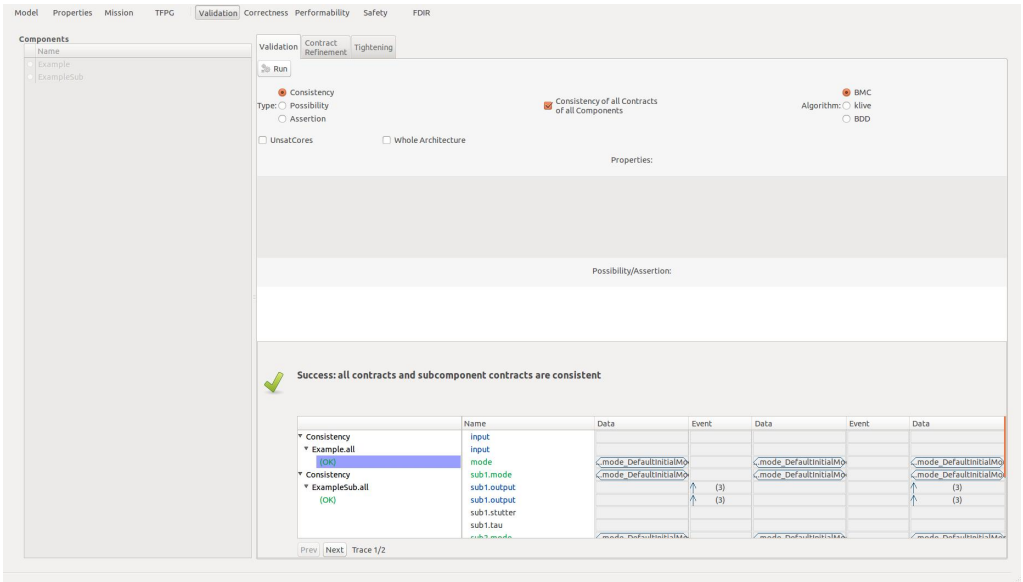


Figure 9.2: Validation - Consistency check

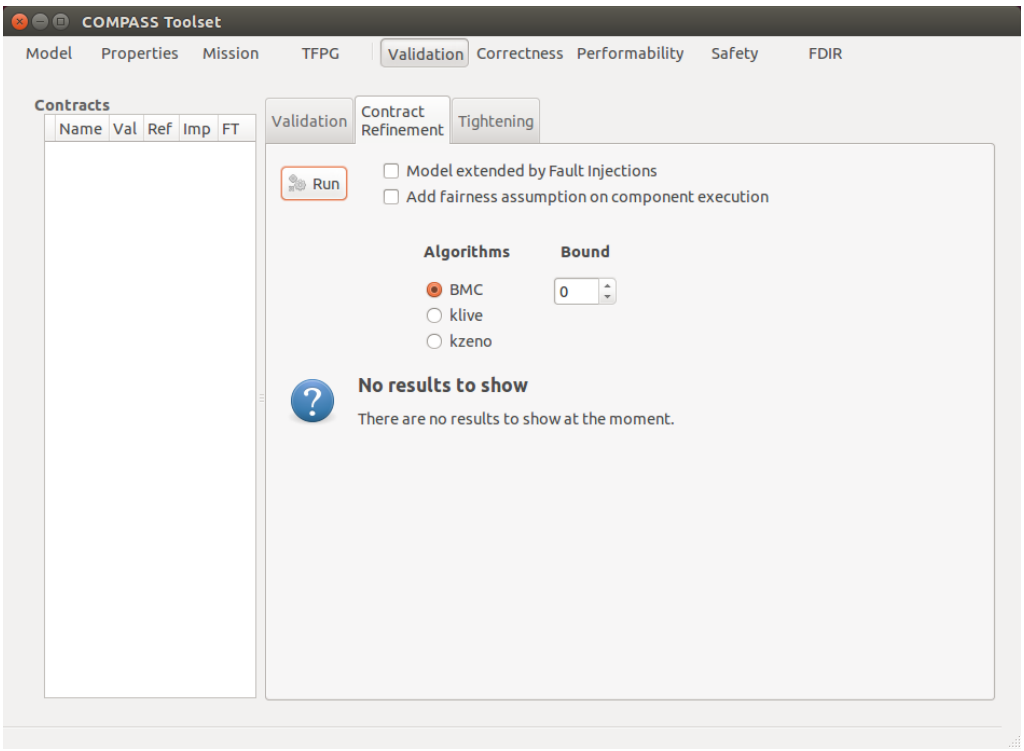


Figure 9.3: Validating contract refinements

9.2.2 Contract Refinement

The Contract Refinement panel, see Figure 9.3, allows the contract refinements specified in the model to be validated. On the left hand side, the contract to be validated can be selected. If *All Contracts* is selected, the validation will be performed for each contract individually.

If fault injections are specified (see Section 6.3), the option *Model extended by Fault Injections* allows them to be enabled or disabled. The option *Add fairness assumption on component execution* allows the fairness assumption to be enabled (see [12]). Under *Algorithms* the analysis algorithm can be selected. For each algorithm, a bound is to be defined under *Bound*.

The result of the validation is indicated in the contract list in the *Ref* column. *OK* indicates the refinements are valid, *BOUND OK* indicates the refinements are valid up to the analyzed bounds, *NOK* indicates a problem with the refinement. Details of the analysis are displayed in the lower half of the panel.

Steps

1. Select the SLIM model from the *Model* pane
2. Select one contract from the *Contracts* pane on the left; this will enable the button *Run*
3. The user can now run the analysis by clicking on the button *Run*, that will fill the pane below with the results of the computation

In example 9.4 contract refinement on `All Contracts` has been performed using algorithm *klive*; a message confirms that everythin is OK.

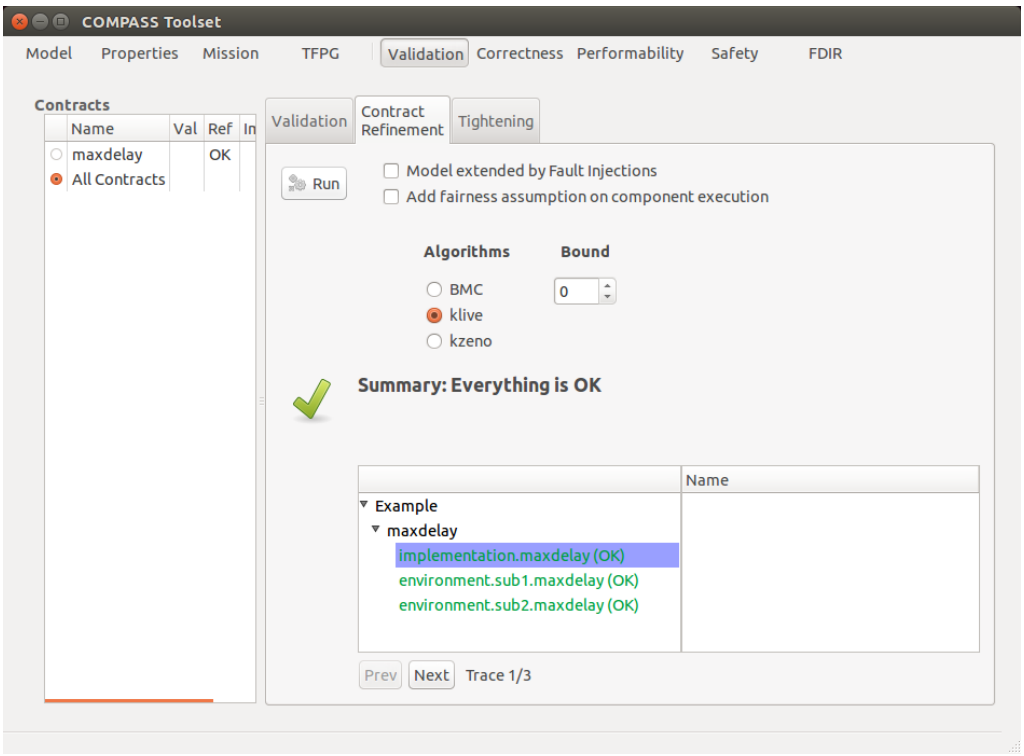


Figure 9.4: Contract Refinement - example

9.2.3 Contract Tightening

The Tightening panel, see Figure 9.5, allows a given contract refinement to be tightened by weakening the assumption of the parent contract and the guarantee of its subcontracts (top-down approach) or strengthening the guarantee of the parent contract and the assumption of its subcontracts (bottom-up approach).

On the left hand side, the component can be selected of which its contracts should be tightened. After selecting the component, the *Contract* list allows the contract to be tightened t to be selected. After doing so, the tightening can be performed.

Two types of tightening are available: The *Top Down* approach does so by weakening the assumptions of any parent contracts and weakening the guarantees of any child contracts. The *Bottom Up* approach does so by strengthening the parent contract guarantees and strengthening the child contract assumptions.

After performing the tightening, the results are shown in the lower part of the panel.

Steps

1. Select the SLIM model from the *Model* pane
2. Select one component from the *Components* pane on the left
3. Select one contract from the *Contracts* pane; this will enable the button *Run*
4. The user can now run the analysis by clicking on the button *Run*, that will fill the pane below with the results of the computation

Figure 9.6 is an example of a result of a (top-down) tightening check.

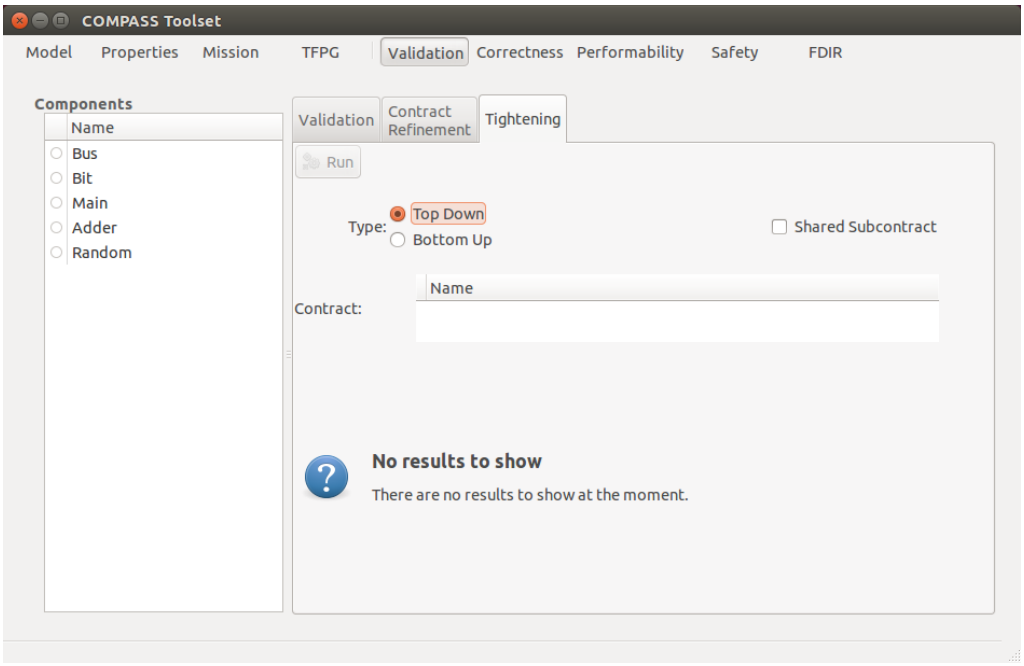


Figure 9.5: Tightening a contract specification

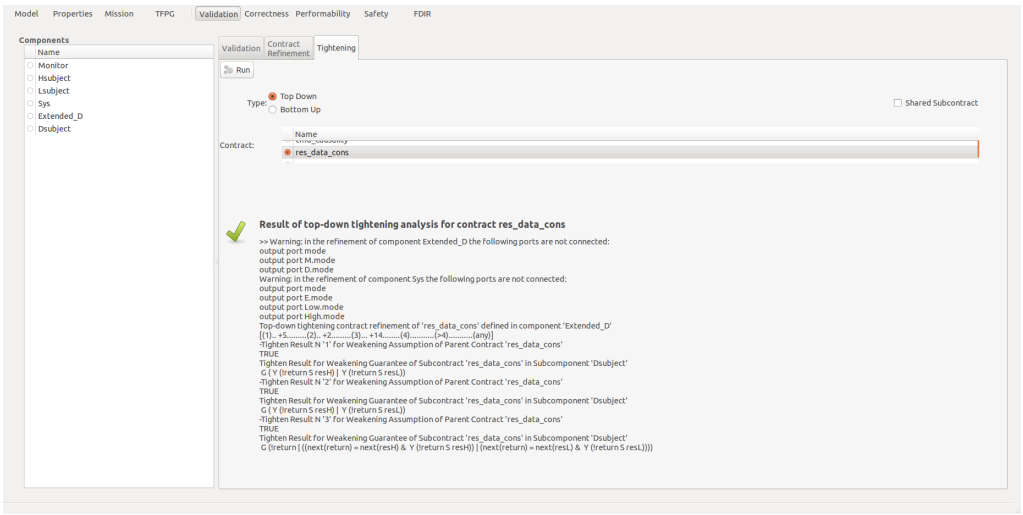


Figure 9.6: Tightening - example

9.3 TFPG

The *TFPG* window gives the user the possibility to perform some model-based reasoning tasks on TFPGs. Three types of analysis are possible:

1. Behavioral Validation: Check if a TFPG is complete or incomplete with respect to a model
2. Synthesis: Synthesize automatically a complete TFPG, starting from a system model and a set of TFPG associations
3. Effectiveness Validation: Check whether a *Failure Mode* of the TFPG is diagnosable or not in a particular *System Mode*.

Before introducing these tabse, we explain what a TFPG is.

9.3.1 Introduction to TFPGs

Time Failure Propagation Graphs (TFPG) are introduce to support various aspects of diagnosis and prognosis in complex systems. They are able to model, for example, the temporal dependency between the occurrence of events of interest, and their dependence on system modes.

A **TFPG** is a labeled directed graph where nodes represent either fault modes, which are fault causes, or discrepancies, which are off-nominal conditions that are effects of failure modes. There are two types of discrepancies: *AND discrepancies* and *OR discrepancies*.

Edges between nodes in the graph capture the effect of failure propagation over time in the underlying dynamic system and they propagate in a time interval *[tmin - tmax]*. Edges in the graph model can be activated or deactivated depending on a set of possible operation modes of the system (*activation modes*); this allows to represent failure propagation in multi-mode (switching) systems.

Figure 9.7 shows how a TFPG is rendered in the GUI in the *Viewer*, which is the first subview that appears when a TFPG is loaded. It is possible to zoom in and out the graph, to move it and to reset all to the inital position. Additionally, by pressing the *Edit* button a textual editor showing a textual representation of the TFPG is opened, by which it is possible to change some aspects of the selected graph.

If errors are encountered while parsing or editing a TFPG, the GUI switches to *Errors* pane, where the error is reported, as shown in Figure 9.8.

All TFPG analysis are carried out by specifying some *TFPG SLIM associations*. **TFPG associations** define TFPG elements (failure modes or discrepancies) in terms of SLIM propositional expression, which can be written and edited in the *Slim associations* pane (see Figure 9.9). In particular it is possible to:

1. Associate failure modes in the TFPG to errors that are currently injected in the SLIM model
2. Associate monitored discrepancies in the TFPG to observable SLIM expressions
3. Associate non-monitored discrepancies in the TFPG to SLIM expressions
4. Assoicate TFPG modes to observable SLIM expressions

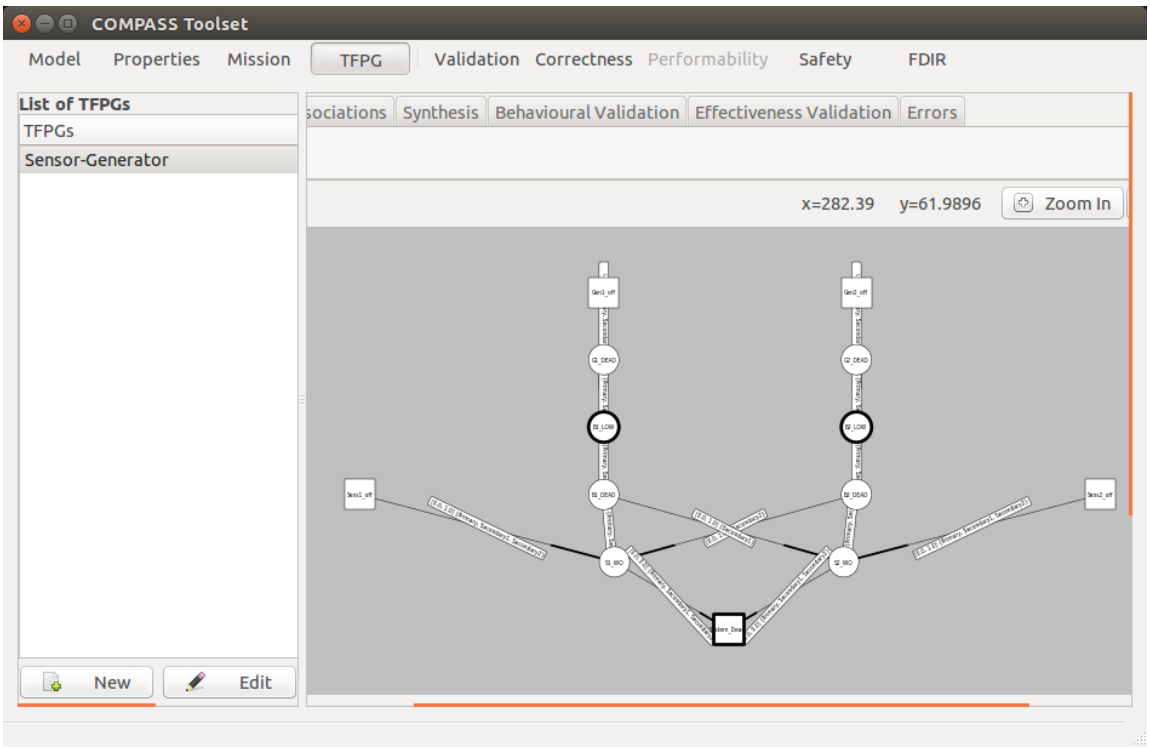


Figure 9.7: Viewing a TFPG

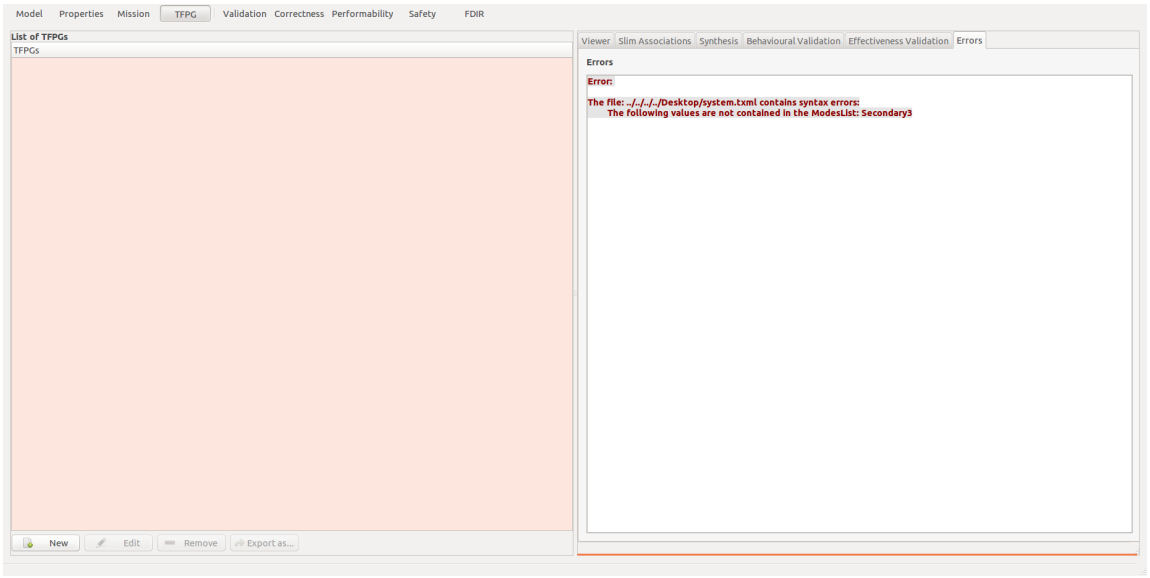


Figure 9.8: TFPG Errors panel

9.3.2 Behavioral Validation

The *Behavioral Validation* panel, see Figure 9.10, allows to check if a TFPG is complete or incomplete with respect to a model; this means to check whether every trace in the system model has a corresponding trace in the TFPG.

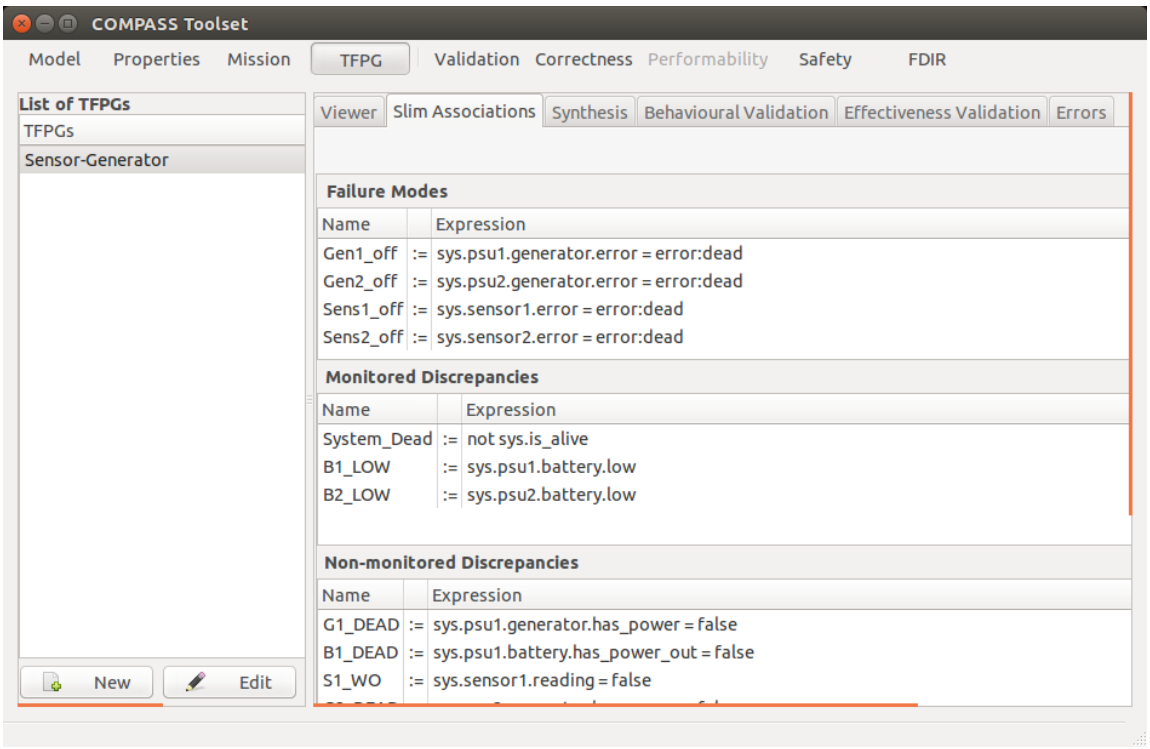


Figure 9.9: Slim associations

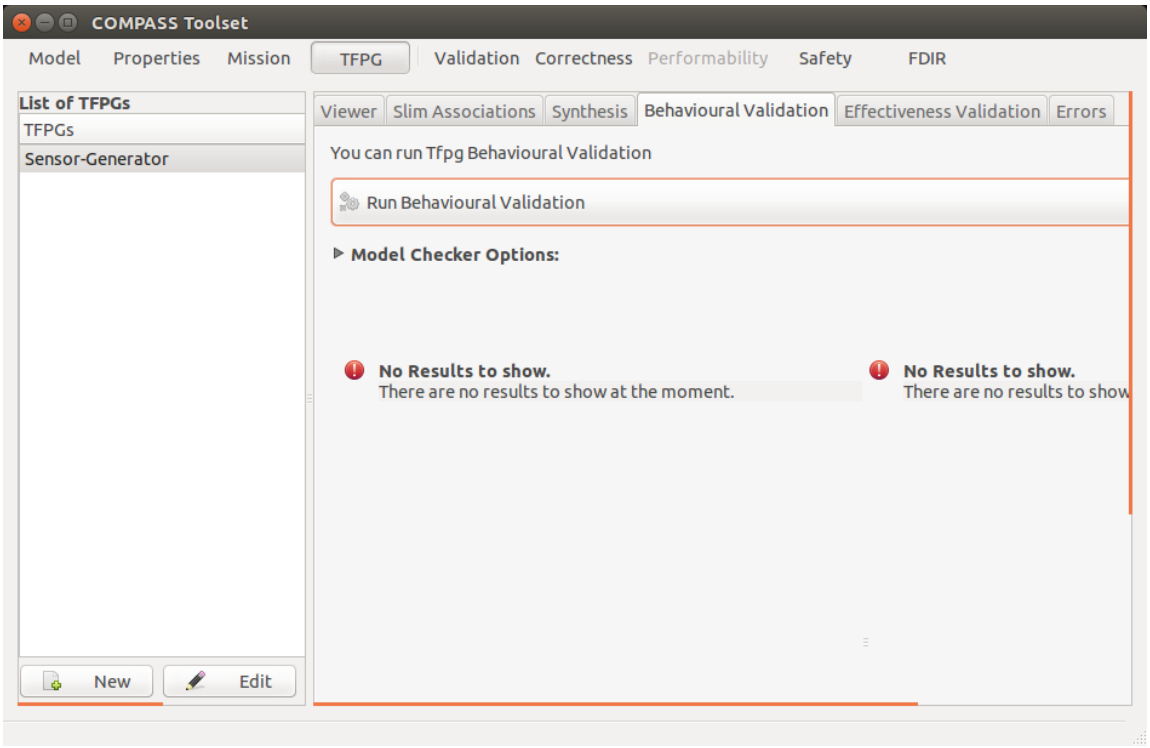


Figure 9.10: Behavioral Validation

Steps

1. Select the SLIM model from the *Model* pane

2. On the left hand side select a TFPG (previously loaded or created)
3. Load slim associations in the *Slim associations* pane; this will enable the button *Run*
4. Optionally, modify the value of the SAT bound (by default 10) in order to refine the check.
5. The user can now run the analysis by clicking on the button *Run*; the results are shown in the lower part of the panel.

Figure 9.11 shows an example of complete TFPG.

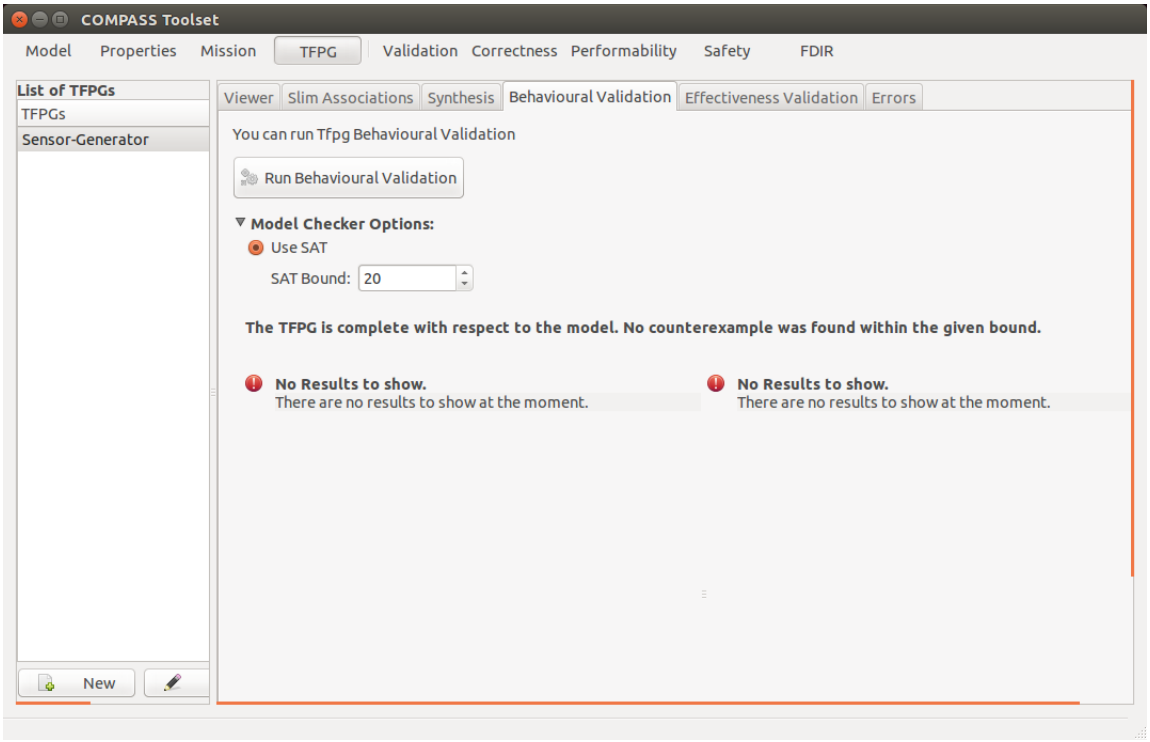


Figure 9.11: Behavioral Validation - Complete result

9.3.3 Synthesis

The *Synthesis* panel, see Figure 9.12, allows to automatically synthesize a complete TFPG, starting from a system model and a set of TFPG associations, which must be specified in the proper part (at least one failure mode, one discrepancy, monitored or unmonitored, and a TFPG mode must be present). After completing the analysis, if has been possible to synthesize a TFPG, it is loaded in the left hand side, otherwise a message is shown to the user in the lower part of the panel.

Steps

1. Select the SLIM model from the *Model* pane
2. Load slim associations in the *Synthesis* pane; this will enable the button *Run*

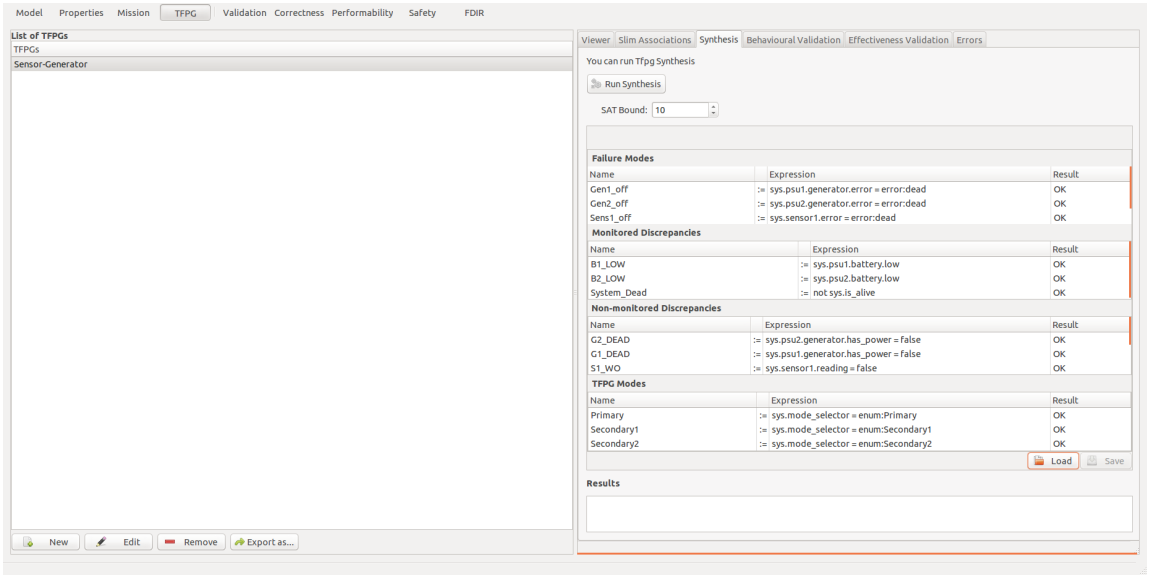


Figure 9.12: Synthesis

3. Optionally, modify the value of the SAT bound (by default 10) in order to refine the check.

4. The user can now run the analysis by clicking on the button *Run*; the results are shown in the lower part of the panel.

In this example (see 9.13) it has been possible to synthesize a TFPG called `Dependency Graph`.

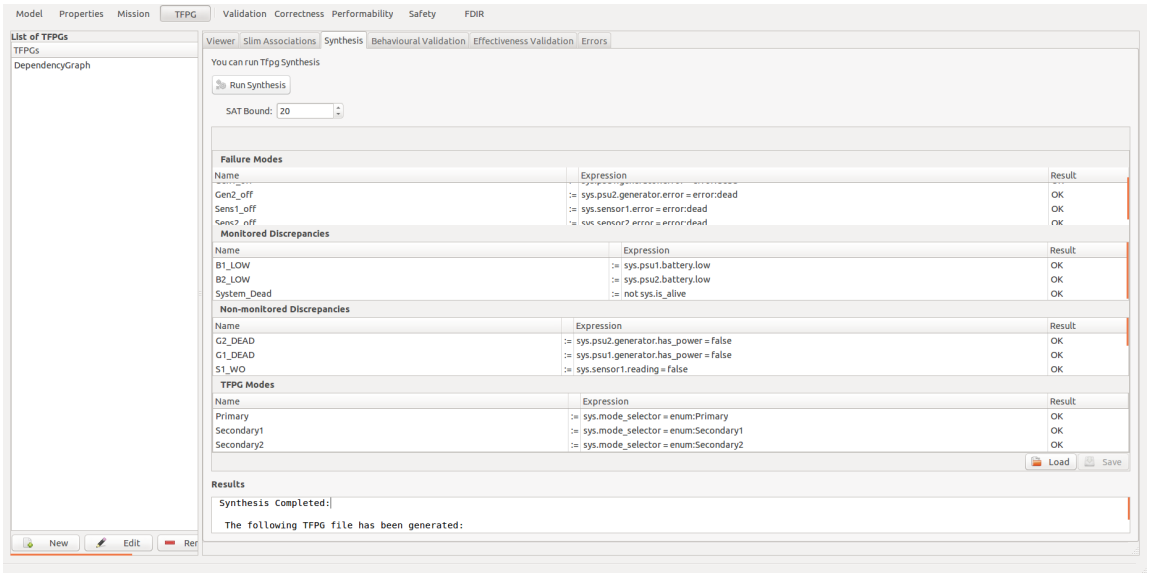


Figure 9.13: Synthesis of a TFPG

9.3.4 Effectiveness Validation

The *Effectiveness Validation* panel, see Figure 9.14, allows to check whether a failure mode of the TFPG is diagnosable or not in a particular system mode. The analysis loops over each pair *Failure Mode, System Mode*, showing the result in real-time; if the pair is found to be not diagnosable, a message appears with the counterexample, otherwise only a message appears. If a counterexample is generated, it shows a pair of traces that witness non-diagnosability for the given failure mode in the given system mode.

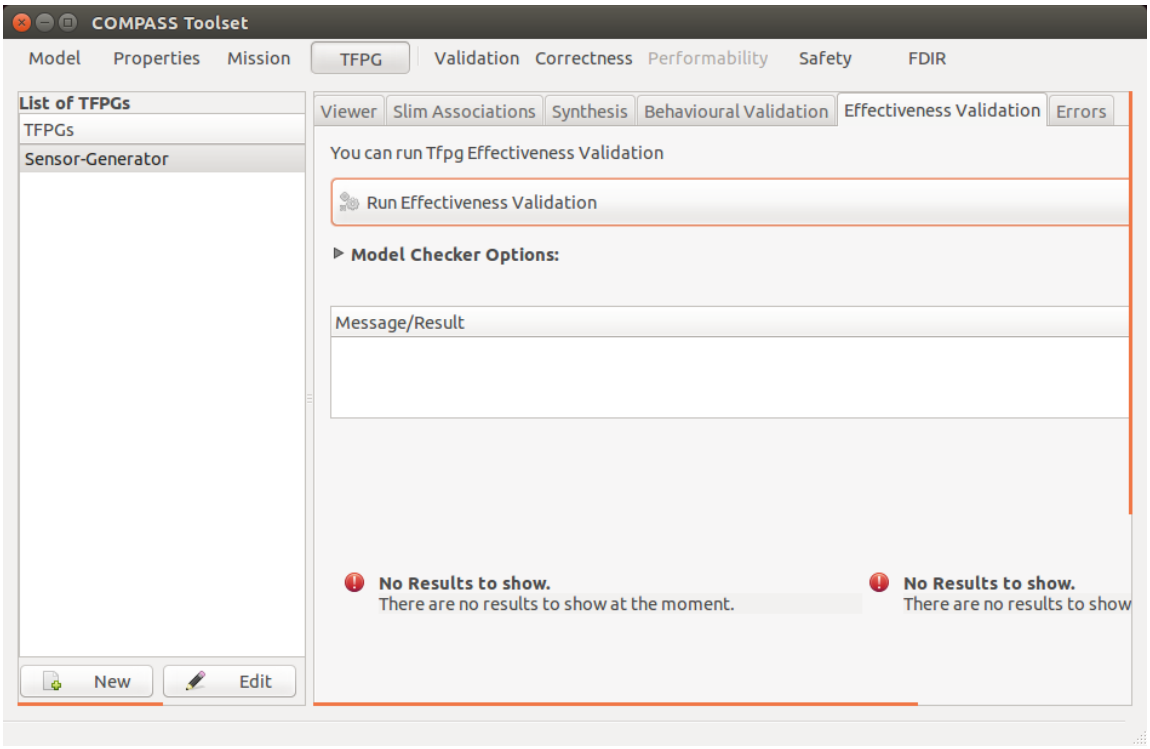


Figure 9.14: Effectiveness Validation

Steps

1. Select the SLIM model from the *Model* pane
2. On the left hand side select a TFPG (previously loaded or created); this will enable the button *Run*
3. Optionally, modify the settings of the engine (*BDD* or *SAT* to refine the check
4. The user can now run the analysis by clicking on the button *Run*; the results are shown in real-time in the upper part of the panel.

In this example (see 9.15), the failure mode `Gen1_off` has been found to be diagnosable in modes `Primary` and `Secondary1`, while it is not diagnosable in mode `Secondary2`; in this case, the counterexample is shown in the lower part of the panel.

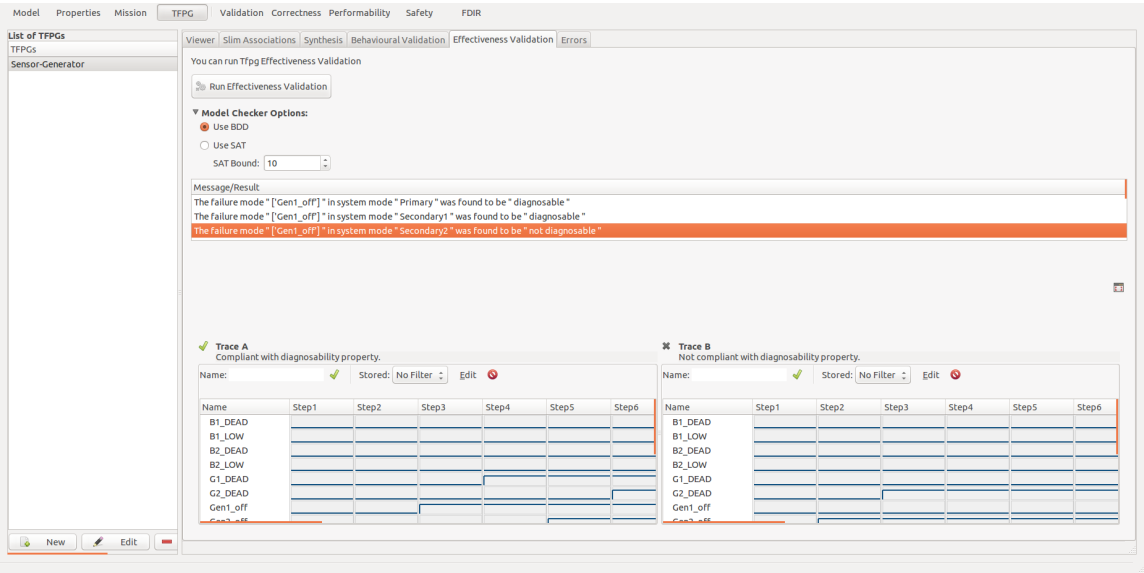


Figure 9.15: Effectiveness Validation - Example

9.4 Verifying Functional Correctness

The *Correctness* window gives the user the possibility to analyze the behavior of a SLIM model, and check if it satisfies a set of desired properties (showing a reason for unsatisfaction, in case the properties do not hold). In addition, as a preliminary sanity check, it is also possible to check the model for absence of deadlocks.

These features are provided by six sub-panes, namely:

1. the *simulation* pane, which allows the user to perform random and guided simulations on the model,
2. the *deadlock checking* pane, which allows the user to verify the absence of deadlocks, and
3. the *model checking* pane, which allows the user to check whether a property holds or not in the model.
4. the *zeno analysis* pane, which allows the user to check if a mode of the SLIM model is Zeno.
5. the *time divergence analysis* pane, which allows the user to check whether a clock is divergent or not
6. the *contract-based verification* pane, which allows the user to check the monolithical verification of a contract

The correctness window is depicted in Figure 9.16. In the following sections we describe the correctness sub-panes in detail. In order to understand the sections 9.4.2 and 9.4.4 it is also recommended to have a look at the section 9.4.1.

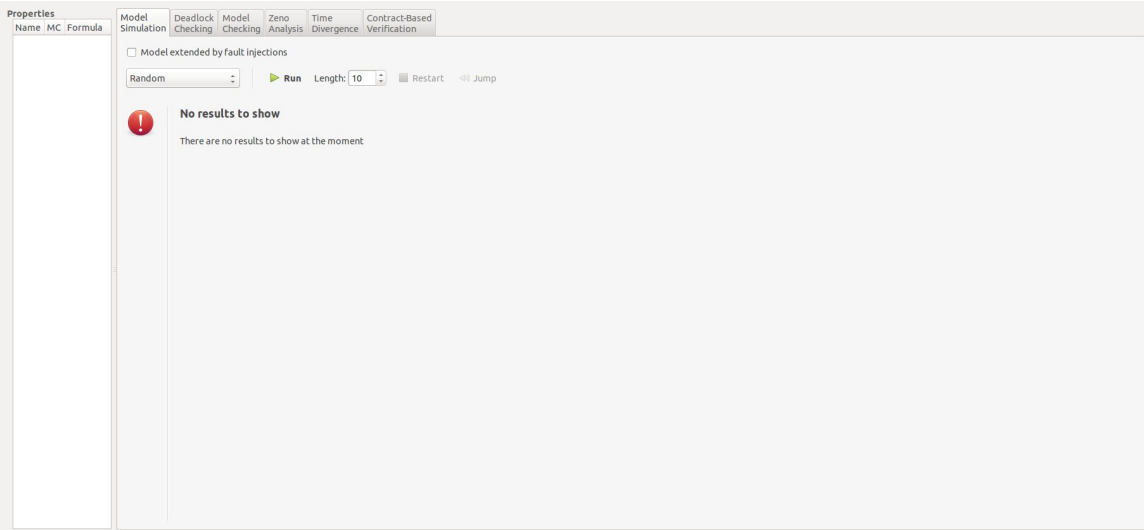


Figure 9.16: The main view of the correctness window

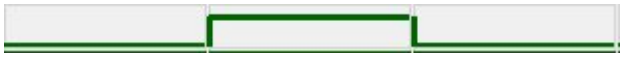
9.4.1 Trace Inspection

A **trace** is a *set of stripes* which describes a particular behavior of the system, they are used in model checking 9.4.4 to provide a counter-example for false properties, and in model simulation 9.4.2 to check the witness of the system.

A **stripe** represents a signal that changes in a discrete (or continuous) timed environment. Each stripe is composed by a name, and a set of values, one for each system's step (i.e. configuration). The steps are linked together by *transitions*, that can be discrete or continuous, and represents the evolution of the system.

From a practical point of view a trace can be viewed as a table, in which rows are the stripes, and columns are the steps.

(a) How the rows underline value changes



(b) How the rows show the Boolean values

Figure 9.17: Trace's details

The notations used in the tables that represents a trace follows:

1. Each stripe (row) have a *type*. Each types has a color associated with, for example blue, and red. In the error stripes the name's foreground is red, and additionally the background of each step is red.

2. When a transition changes a value in a stripe, his cell looks different in order to remark the variation (see Figure 9.17(a)).

3. if the values are Boolean, than a square wave is shown (see Figure 9.17(b)).

4. Finally the continuous transitions are underlined by a magenta background on the steps that derives from them.

Filtering features

Another important feature of the trace, is the trace filtering.

The trace filters can be accessed through a trace filter bar at the bottom of the trace (for more information about the trace read the paragraph below). Using filters, it is possible to filter the set of signals that will be shown in the window. It is possible to filter depending on the name, the type, and the number of steps of the trace. The procedure to use the filter bar is the following:

1. The filter bar (compare Figure 9.18), is composed by a 'quick' part in which the user can apply a quick filter (only on the names), and a 'stored' part in which advanced filtering options are enabled. In the latter case, the filters will be automatically stored.

2. Figure 9.19 shows an example of usage of the quick filter (on the fly filtering on the names). The filtering on names supports the full power of the regular expressions, so any regular expression can be used.

3. In order to use the full features of the filters (i.e., filtering on steps, on types and on names) the user has to create a "stored filter, by clicking on the *Edit* button in the filter bar. This opens a pop-up window (shown in Figure 9.20) that allows more complex filtering. The filtering on names was explained before, the filtering on the steps can be done writing in the *Filtered steps* field a sequence of comma separated ranges, each with the following syntax:

   (a) *a single number*, that represents a single step that will be shown when the filter is active (i.e. a single number like 7 or 8)

   (b) or *a two side bounded range*, that represents a set of steps that will be shown when the filter is active. This range is composed by a starting number, which is the starting step, and a special symbol '-' that represents the range, and an end number, which is the ending step. (i.e. 7-19 shows all the steps from 7 to 19, both included)

   (c) or *a single side bounded range*, that represents an unbounded set of steps that will be shown when the filter is active. This range is composed **optionally** by a starting point (the same as the starting point in the two side bounded range), a special symbol '-' that represents the range, and **optionally** by an ending point (the same as the ending point in the two side bounded range). The missing bound means *until there are steps shows them*, (from here to the begin if the the starting bound is not specified or from here to the end if the ending bound is not specified). At least one of the two bound must be set, in order to give meaning to the filter. (i.e. 9- means from 9 to the end of the trace, -5 means from the step 1 (that is the first step) to the step 5).

If one or more ranges are not valid, the button *Ok* will be automatically disabled. Finally the *Filtered types* entry can be a list of comma separated types that have to be shown. The supported types are 'input', 'type', 'combinatorial' and 'error'.

Figure 9.18: The filter bar

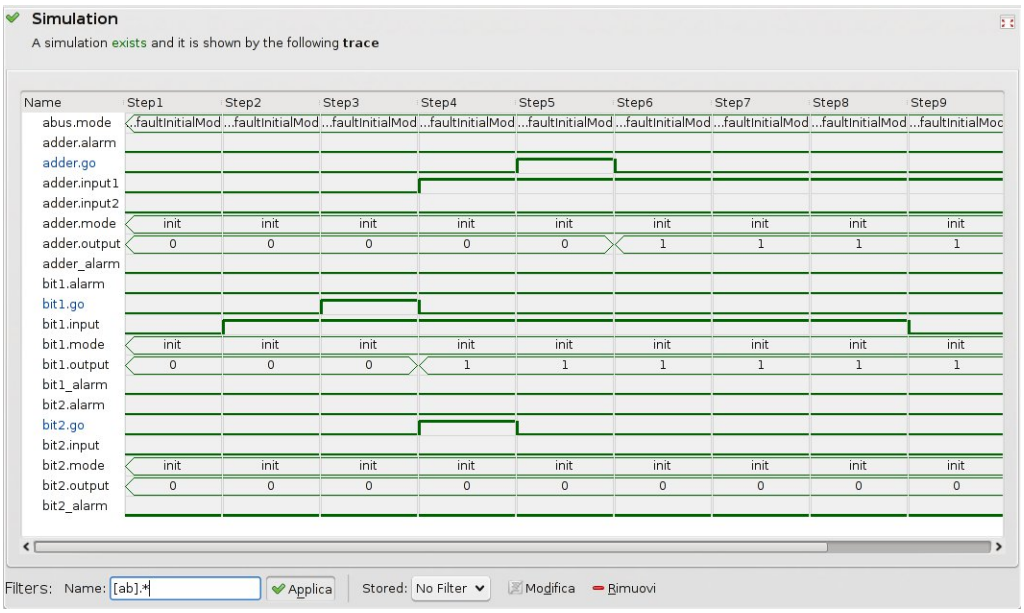


Figure 9.19: Quick filtering

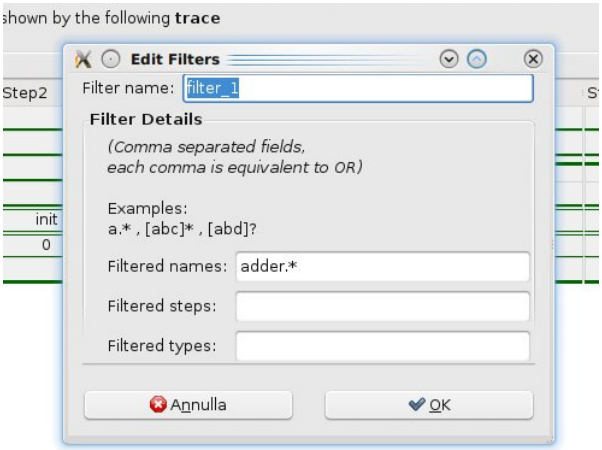


Figure 9.20: Advanced filtering

The button *Remove* is used to delete the current active stored filter (if there are no active filters, then clicking on the this button has no effect).

A quick essay of the syntax of regular expressions is presented in the following table:

Char	Meaning	Example
.	Matches any character except a newline	`a.b` matches aab, abc, acb, etc.
∧	Matches the start of the string	
$	Matches the end of the string	
⋆	Matches 0 or more repetitions of the preceding RE	ab* will match a, ab, or a followed by any number of b.
+	Matches 1 or more repetitions of the preceding RE	ab+ will match a followed by any non-zero number of b; it will not match just a.
?	Matches 0 or 1 repetitions of the preceding RE	ab? will match either a or ab.
\	Escapes special characters (permitting you to match characters like '*', '?', '.', and so forth)	a\.b will will match 'a.b'
[]	Used to indicate a set of characters. Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a '-'.	For example, [akm$] will match any of the characters 'a', 'k', 'm', or '$'; [a-z] will match any lowercase letter, and [a-zA-Z0-9] matches any letter or digit.
A|B	where A and B can be arbitrary REs, creates a regular expression that will match either A or B. An arbitrary number of REs can be separated by the '|' in this way.	

9.4.2 Model Simulation

The *model simulation* pane is the first one. It is typically an useful starting point to check the behavior of a model. It allows the user to carry out three activities:

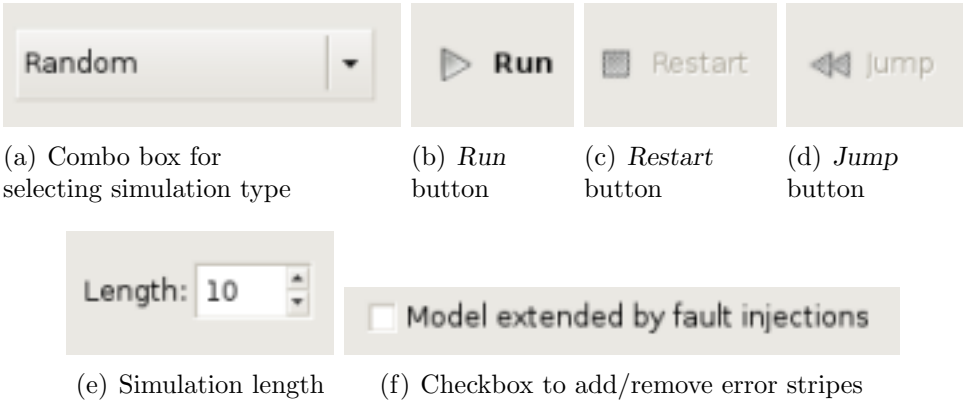
- *random* simulation, in which the values of the model signals are chosen automatically by the system and
- *guided by transitions* simulation, in which the user can explicitly force the system's behavior by picking one of the available transitions.
- *guided by values* simulation, in which the values of the model signals can be forced by user in order to verify the behavior of the system under specific scenarios.

During one simulation, it is possible to switch among the different types at any moment.

The simulation pane, shown in Figure 9.21, is shared among the three simulation types, and contains control buttons.

From left to right, the following items are shown:

Figure 9.21: The main control buttons of the model simulation



(a) Combo box for selecting simulation type

(b) *Run* button

(c) *Restart* button

(d) *Jump* button

(e) Simulation length

(f) Checkbox to add/remove error stripes

Figure 9.22: Detailed descriptions of the simulation pane control buttons

1. A combo box (initially labeled *Random* that can be used to set the type of the simulation that the user wants to carry out (compare Figure 9.22(a)).

   Other available values are *Guided by transitions* and *Guided by values*.

2. A button *Run* that starts the simulation and causes an update of the trace viewer table (compare Figure 9.22(b)).

3. A button *Restart* that is available only when there is an active simulation, (i.e., the button *Run* was clicked) and restarts the simulation (i.e., it goes back to the first step). Note that in this case, all the other steps are lost (compare Figure 9.22(c)).

4. A button *Jump* that is available only when there is an active simulation (i.e., the button *Run* was clicked) allows the user to go back to a previous step by clicking first on the button *Jump* and then on the column labeled with the step number to which the user wants to go (compare Figure 9.22(d)).

5. An option field that allows the user to choose how many steps will be automatically carried out (both in random and in guided simulation) by the simulator (compare Figure 9.22(e)).

6. Finally, the check box labeled *Simulate the model extended by fault injections* allows the user to simulate a SLIM model extended with the specified fault injections. If the check box status changes when a trace is active, the next simulation will start from the beginning (state 1) in order to add (if the check box is selected) or remove (if the check box is not selected) the stripes that represents the fault injections (see Figure 9.22(f)).

In the next sections we explain in more detail how the traces generated by *random simulation*, *guided-by-transitions* and *guided-by-values simulation* look like.

Random Simulation

In random simulation the trace appears after the first click on the *Run* button, below the controls of the model simulation pane, and looks like in Figure 9.23.

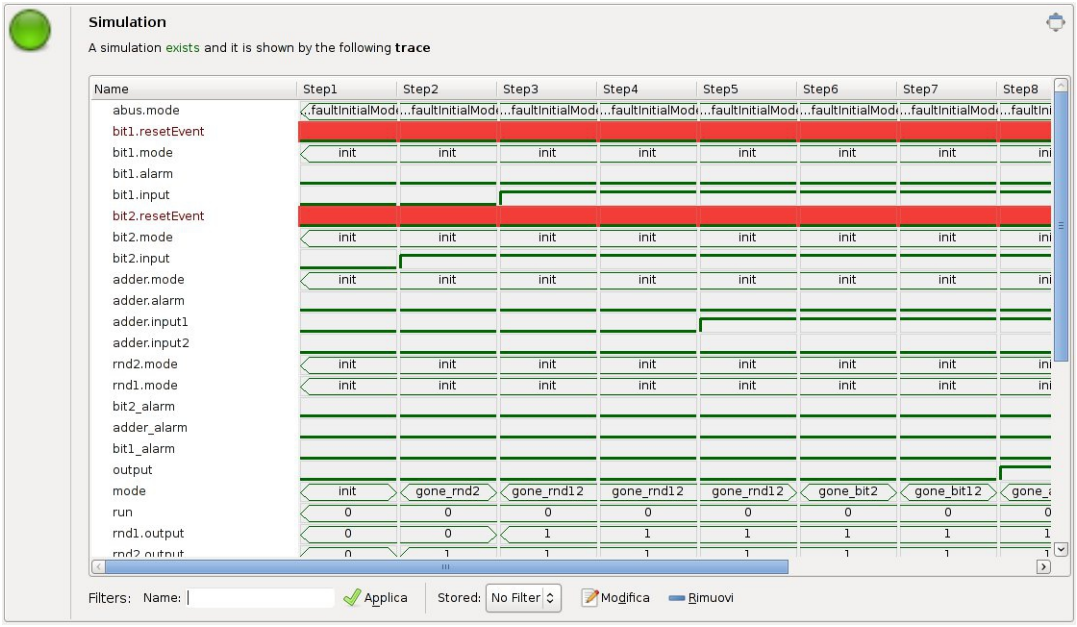


Figure 9.23: A simulation example

As we can see the trace is composed by a table where the first column provides the name of a model element, and in the other columns the value of that element, for each step of the simulation, is shown. The fault injection stripes are shown with a red background, whereas the Boolean stripes are shown as a waveform with two values, the "high" one for the true value, and the "low" one for the false value.

Moreover, it is also possible to use the filtering features that are explained in Section 9.4.2.

Guided-by-Transitions Simulation

In Guided-by-Transitions simulation it is possible to select step after step either single discrete transitions to be executed in the simulation, or time-transitions if the currently loaded SLIM model features timed/hybrid behaviors.

Figure 9.24 shows where the type of guided transition (Discrete vs Timed) can be chosen. In guided *Discrete* transitions simulation, the simulation pane is split in three sections:

1. the left-top section consists of a tree viewer which lets the user choose among available implementations for the model which is currently being simulated. Here the number of enabled transition is also indicated.

2. the left-bottom section lists available transitions for the implementation selected in the above section. The unavailable transitions are struck through and not selectable (but still visible). Enabled transitions are instead selectable.

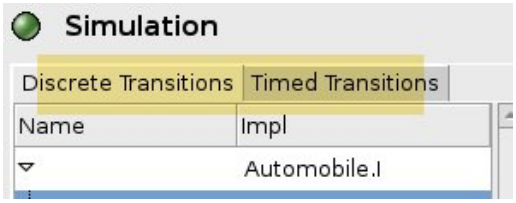


Figure 9.24: Guided-by-Transitions types

3. the right section contains the visual representation of the trace, similarly to random simulation as described above.

An example of discrete transitions guided simulation is shown in Figure 9.25.

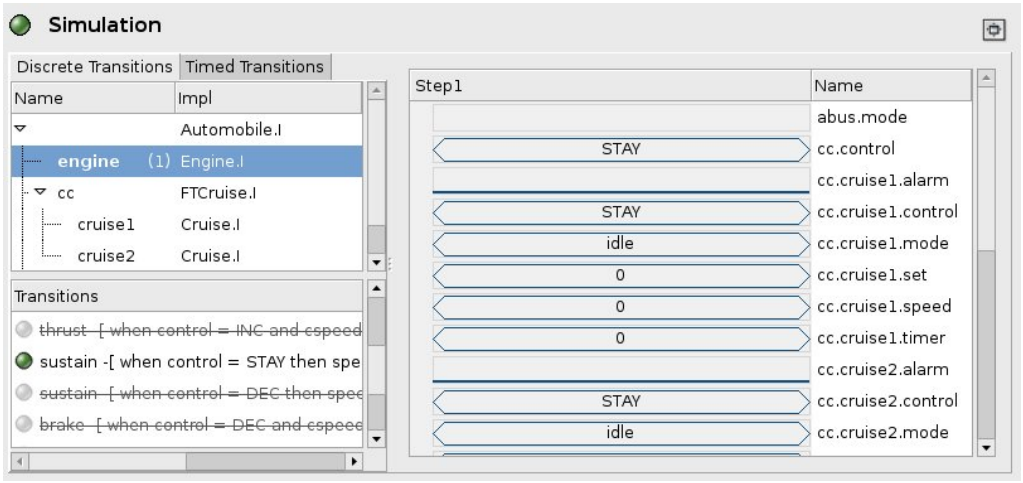


Figure 9.25: Discrete-transitions guided simulation

To carry out a single discrete transition, it is needed to:

1. Select one Component implementation containing at least one enabled discrete transition.

2. Select the transition among those which are enabled

3. Click the 'Run' button in the simulation pane, or alternatively double-click the transition chosen at the previous step.

When a single transition is carried out, the shown trace gets extended by one step, and the simulation can be continued on.

If the currently loaded SLIM model contains hybrid/timed behaviours, and when it is possible to carry out a timed-transition, the pane for guided *Timed* transitions simulation is available.

The Timed-transitions guided simulation pane is shown in Figure 9.26. When performing a time-transition, it is possible to specify some constraints about the time duration of the transition. For example in Figure 9.26 duration has to be greater than 0.0 and lesser than or equal to 5.0 time units.

To carry out a single time transition, it is needed to:

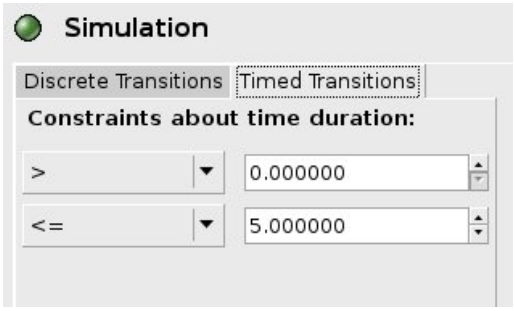


Figure 9.26: Timed-transitions guided simulation, with some constraints specified

1. Select the Timed Transition pane.
2. Optionally specify constraints for the time duration.
3. Click the 'Run' button in the simulation pane. The button may be not enabled when the optionally specified constraints are invalid or inconsistent with the current model state.

Figure 9.27: After a Timed-transitions has been carried out

When a time-transition is carried out, the shown trace gets extended by one continuous step, and the simulation can be continued on. However, the next transition will have to be a *discrete* transition, as two consecutive timed transitions are not allowed to occur by construction. Figure 9.27 shows the result of a timed-transition with the mentioned constraints.

Guided-by-values Simulation

In guided-by-values simulation, the trace is shown differently with respect to random simulation. There are two representations of the same trace, namely:

1. the left one shows the last two steps plus an additional one that is editable, so the user can **force** the future value of the simulation in order to see how the system behaves
2. the right one shows the complete trace (with all the steps) but the stripe name is at the end.

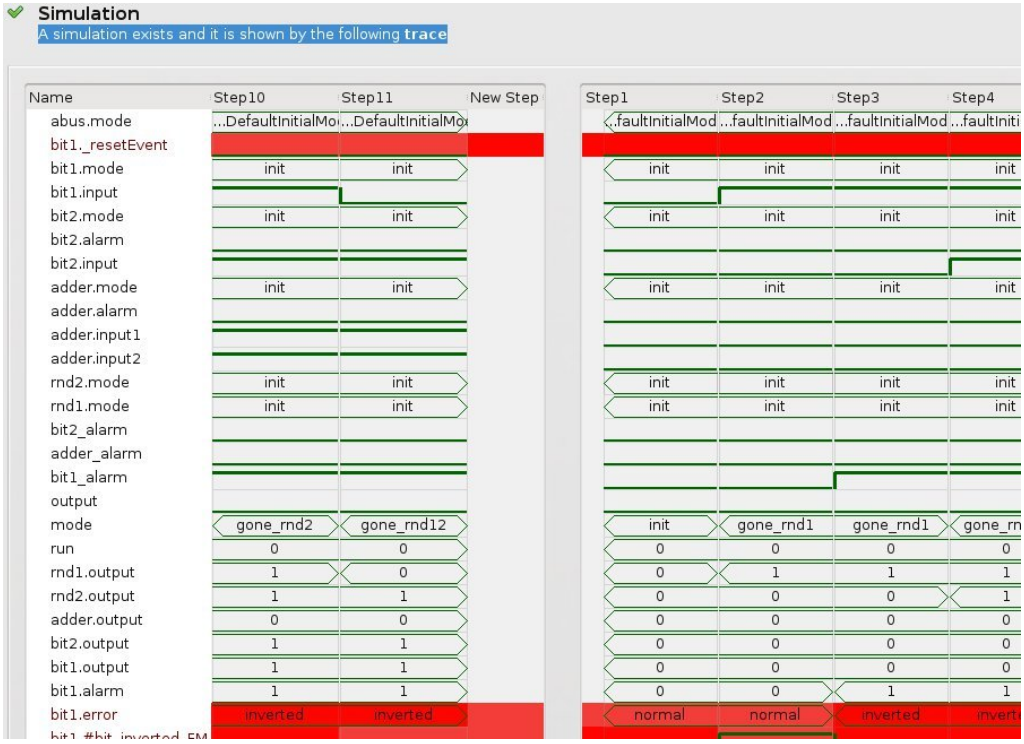


Figure 9.28: Guided-by-values simulation

An example of guided-by-values simulation is shown in Figure 9.28.

In order to force values in the stripes, the user has to follow this procedure:

1. Click on the *new step* column in the row that represents the stripe to be edited.
2. Insert the value to be forced. If the value is OK, the background of the cell becomes green, otherwise the background becomes red, and a tooltip will be shown with a description of the error.

If no values are forced, the simulation behavior does not change. Otherwise, the system try to simulate the system forcing the constraints specified by the user, namely, if the trace generated for a given integer stripe (that represents the evolution of a signal) ends at the step 9, with value '0', and in the *new step* column, in the given stripe line the users write '1', the system at the next simulation try to force the value of the given integer signal to be '1'.

If this operations fail, no steps will be added to the trace, and an error message is shown (compare Figure 9.29).

Moreover, it is also possible to use the filtering features explained in Section 9.4.2.

9.4.3 Deadlock Checking

In the deadlock checking pane, the user is presented with a button labeled *Run Deadlock Checking*. By pressing this button, the current SLIM model is checked for the presence of possible deadlocks. The corresponding pane is shown in Figure 9.30.

There are two possible outcomes of this analysis:

1. When the model does not contain deadlocks, the message in Figure 9.31 is shown.

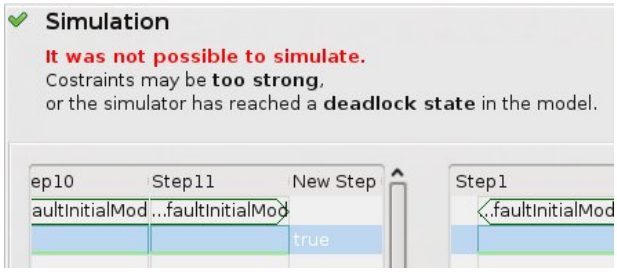


Figure 9.29: Guided-by-values simulation failure example

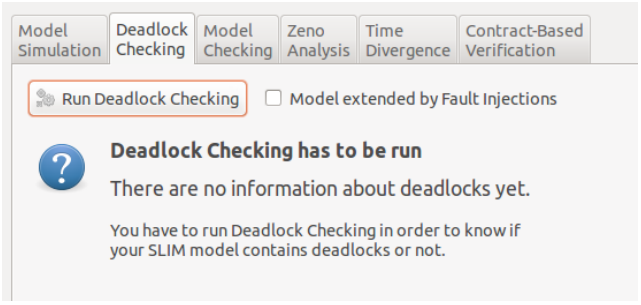


Figure 9.30: The deadlock checking pane

2. Otherwise, the message in Figure 9.32 is shown.

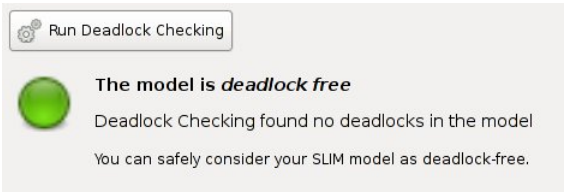


Figure 9.31: Model deadlock free

Note that deadlock checking is disabled when the model is hybrid (see section 9.1).

9.4.4 Model Checking

The *model checking* pane is used to check a SLIM model against one or more properties. The properties must be chosen among the available ones. The initial window looks like in Figure 9.33.

The underlying model checker (NuSMV) allows for model checking of both finite purely discrete and infinite hybrid/timed SLIM models. In the case of finite discrete models, two techniques are available: BDD-based model checking and SAT-based model checking. In the case of infinite hybrid/timed models, only SAT-based model checking is available.

Figure 9.34 show the options which are provided to select the underlying technology to be used by the model checker.

In the following, a short description of these techniques is given, along with the main advantages and drawbacks each of them offers. Some practical guidelines are also provided, in order to make the user aware of which options are available and what she may expect to encounter especially when tackling big models.

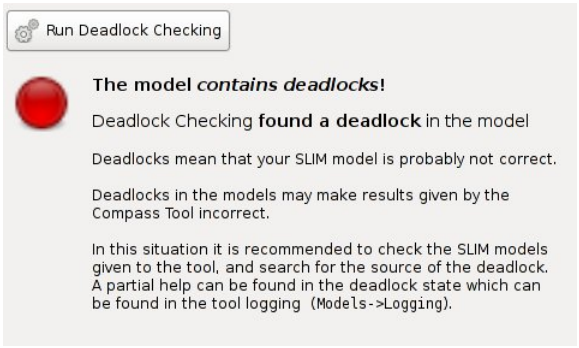


Figure 9.32: A deadlock found in the model

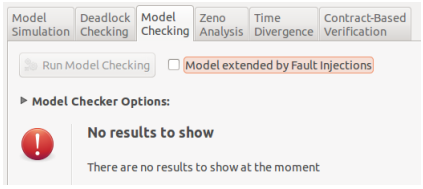


Figure 9.33: The model checking initial window

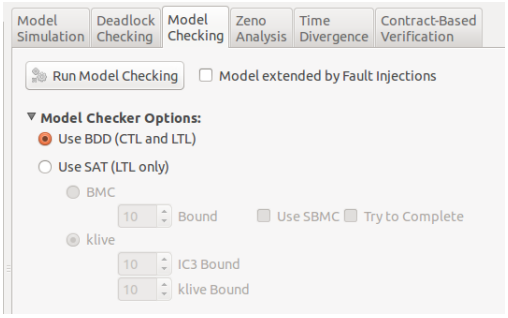


Figure 9.34: Model checking options

BDD-based Model Checking

BDDs (Binary Decision Diagrams) are efficient structures used to represent a set of states symbolically. They represent the basement structure used by NuSMV when model checking properties. The model checking algorithm consists of an exhaustive search of the state space of the concurrent system to determine truth of property. The search is *complete*, and the result is either "True" if the property holds, or "False" if it does not. If the property does not hold, a counter-example is also extracted and shown to the user.

However, it is important to consider that the length of the provided counter-example is not guaranteed to be minimal, which is instead the case when using SAT-based model checking.

Even if BDDs represent states symbolically (and not explicitly like in other techniques like in explicit-state Model Checking), and even if NuSMV features advanced heuristics for achieving efficiency and partially control the state explosion, BDD-based Model Checking can be a very resource-consuming (both time and memory) activity.

To enable BDD-based model checking, option **`Use BDD`** (see Figure 9.35) has to be selected.

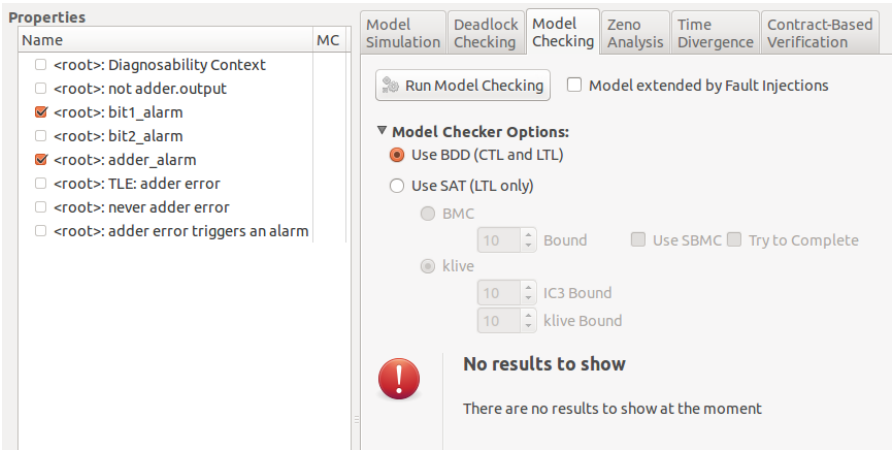


Figure 9.35: The Model Checking pane with two properties selected

SAT-based Model Checking

If from one hand BDDs can provide safe and certain answers, on the other it might be not able to provide them within the available resources. In this case exploiting SAT-based techniques can help.

SAT-based technique is an alternative to BDD-based Model Checking, and can be used when latter is not available (timed/hybrid SLIM models), or when BDDs is too resource-consuming to provide an answer.

SAT exploits a techniques called Bounded Model Checking (BMC). In BMC the property to be verified is encoded in LTL, negated and then manipulated to be combined with the SLIM model (tableau construction).

The combined SLIM model and the negated property's tableau are then "unrolled" linearly into steps with a given *bound*, to produce a *BMC problem*. Each step of the unrolling corresponds to a single discrete or hybrid/timed configuration-transition of the SLIM model, starting from the initial state.

After that a K-bounded BMC problem has been produced, a SAT solver (or a SMT[1] Solver for the timed/hybrid cases) is invoked to search for an execution of the SLIM model which can satisfy the BMC problem. If such execution exists, it is a witness of the violation of the property, the property can be safely told to be "False", and the witness can be presented as a counter-example proving it.

One advantage on BDD-based model checking is that the provided counter-example is assured to be minimal, meaning that there exists no other possible counter-example whose length is lesser than the length of the given counter-example.

Otherwise if no execution can be found, then the property can be safely told to be "True up to step-K".

However, it is not possible to state safely that the property holds. It is then possible to increase K to $K+1$ and search for an execution of the SLIM model which satisfy the $K+1$-bounded BMC problem.

For example with bound 0 the property is checked only against the initial state. If the bound is 1, all the possible states reachable with *any* single transition (discrete or timed) are

[1]Satisfiability Modulo Theory

checked. The search is exhaustive, but it is *bounded*.

If no counterexample is found within the specified bound, the message "The property is true up to bound "K" is shown. (See Figure 9.37

Tackling incompleteness of SAT

BMC is an incomplete method unless a value for the bound K can be determined to guarantee that no counterexample can be missed. There are a few works (see [10] and [6]) which have investigated techniques for computing this bound, but at the moment these techniques have not been implemented, and deciding a sufficient value for the bound is still an open research issue.

However, following the intuition that simpler problems are easier to solve (and the model can then be explored with deeper bounds), a technique called Simple Bounded Model Checking (SBMC) (see [11]) has been implemented in NuSMV.

SBMC presents an encoding for the BMC Problem which is linear in the bound, the system description (i.e. the size of the transition relation as a propositional formula) and the size of the property as an LTL formula. The resulting propositional formula has both a linear number of variables and clauses.

Furthermore, SBMC has been extended with an option which make it a *complete* checking technique. This means that even using SAT technology it is possible to tell the user that a given property is true. However, enabling completeness checking may increase computation times significantly, and it is available only for discrete SLIM models.

If SBMC is not available as the model is timed/hybrid, or if SBMC cannot conclude that a property is true, then it is up to the user[2] to decide which is a sufficient bound for the analysis depth. In this case Model Checking gets closer to standard testing/simulation, with the big difference that in (Bounded) Model Checking *all* possible execution up to a given bound are explored exhaustively, whereas in testing/simulation each execution has to be tested separately.

K-Liveness

K-Liveness [5] is an algorithm to prove liveness properties such as LTL properties by reducing the problem to prove an invariant. The algorithm builds a monitor for a "live" signal so that the original property is satisfied if and only if the signal can be seen only finitely many times. K-Liveness proves therefore the property by checking that such signal can be visited at most K times (so K is the bound used for K-Liveness). Note that, in the case of infinite-state systems, the problem is undecidable and such method is sound but incomplete: there may be systems for which no K is sufficient to prove that the property.

K-Liveness is integrated with an efficient algorithm for checking invariant called IC3 [2]. IC3 is a symbolic model checking algorithm which combines a search-based approach with inductive reasoning. It builds an over-approximation of the reachable state space, using clauses obtained by generalization while disproving candidate counterexamples. COMPASS actually uses IC3IA [3], an extension of IC3 for infinite-state systems, which integrates IC3 with implicit predicate abstraction [13].

When dealing with timed systems K-Liveness is further enhanced with the technique described in [4], which ensures that the time progresses between consecutive occurrence of the

[2]This is indeed a limit of the current technology

"live" signal. Although the algorithm is in general still incomplete, it is quite effective on many practical problems.

Notes on Model Checking Timed (and Hybrid) Specifications

In addition to what explained in the previous section, there are other reasons for incompleteness in the case of timed and hybrid models. In fact, in general it is well known that even 'simple' problems such as reachability analysis are already undecidable for simple classes of timed and hybrid models. This means that there are models for which model checking is not guaranteed to find a counterexample (for any bound) even if one exists. In addition to this, the user must be aware of the following characteristics and limitations of the current technology:

- model checking is carried out on *fair* paths. In the timed and hybrid context, a path is considered fair if timed transitions occur infinitely often. In addition, time must not converge to a finite bound. In particular, paths which show Zeno behaviors or timelocks (compare Section 5.1.1) are excluded. The user must be aware of the fact that for models that do not admit fair paths, the result of model checking universally quantified properties is trivially true (a property is true on all fair paths, as there are no fair paths). Compare Section 5.1.1 for ways to avoid such behaviors.

- the current SMT-based technology for timed and hybrid systems looks for infinite counterexamples that are loop-shaped (finite traces with a 'back-pointer' to a previous state). As such, this technology is not able to find infinite counterexamples that cannot be expressed in this way. A notable example is given by models that show time divergence (compare Section 5.1.1). In such cases, the current technology is not guaranteed to find counterexamples even for theory fragments and problems that are decidable (e.g., reachability for timed automata). We refer to Section 5.1.1 for ways to avoid such behaviors.

Note that these characteristics and limitations also hold for other analyses based on model checking such as Safety and Dependability Analyses. We refer to the COMPASS project final report for an additional discussion about future perspectives in this area.

Options for SAT technology

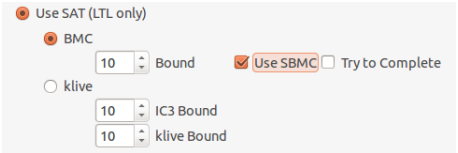


Figure 9.36: The SAT technology options

Figure 9.36 shows the parameters which are available when SAT is enabled. In particular, there are a few options which are listed here:

- **`SAT Bound`** is the maximum number of steps (bound) that SAT will analyze in order to find a counterexample. The search starts from 0 and incrementally goes to the given maximum bound.

  If a counterexample is not found within the specified bound, an the user is warned with a message (see Figure 9.37)

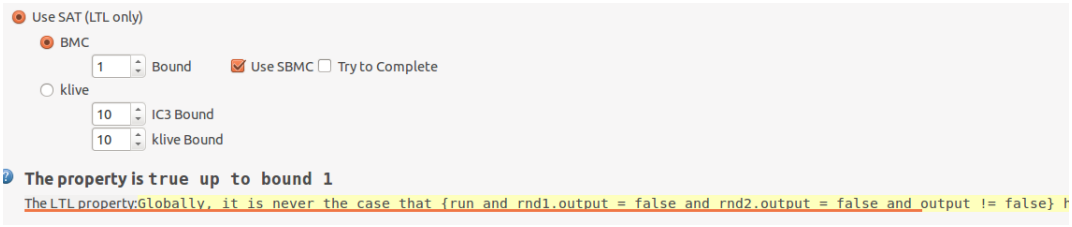


Figure 9.37: SAT cannot tell the user if a property is true

- **`Use SBMC`** enables the model checker to use Simple Bounded Model Checking, which is a (possibly faster) alternative with respect to standard Bounded Model Checking.
- **`Try to Complete`** instructs the solver to use induction techniques to provide conclusive result when a property holds. This option should be used with care, as it may increase computation times significantly.

Running Model Checking and inspecting results

After either a Model Checking technique has been chosen among the available ones, it is possible to run the model checker.

One or more properties has to be selected in the Properties pane (see Figure 9.35).

Clicking the button **`Run Model Checking`** will sequentially invoke the model checker for each selected property, and the truth value of each property is shown by an icon in the **`MC`** column at the end of each computation (see Figure 9.38).

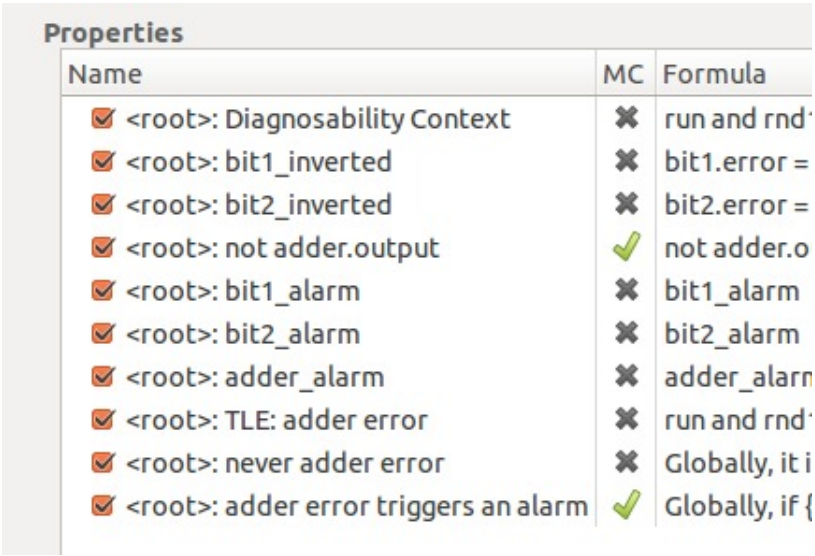


Figure 9.38: The Model Checking pane with some properties model checked.

Double-clicking the icon will show the complete result information about the property which the icon is associated to (see Figure 9.39).

If when clicking the button **`Run Model Checking`** only one property is selected, then the result pane will show the result with no need for double-clicking the icon in the **`MC`** column.

There are two possible outcomes in the result pane:

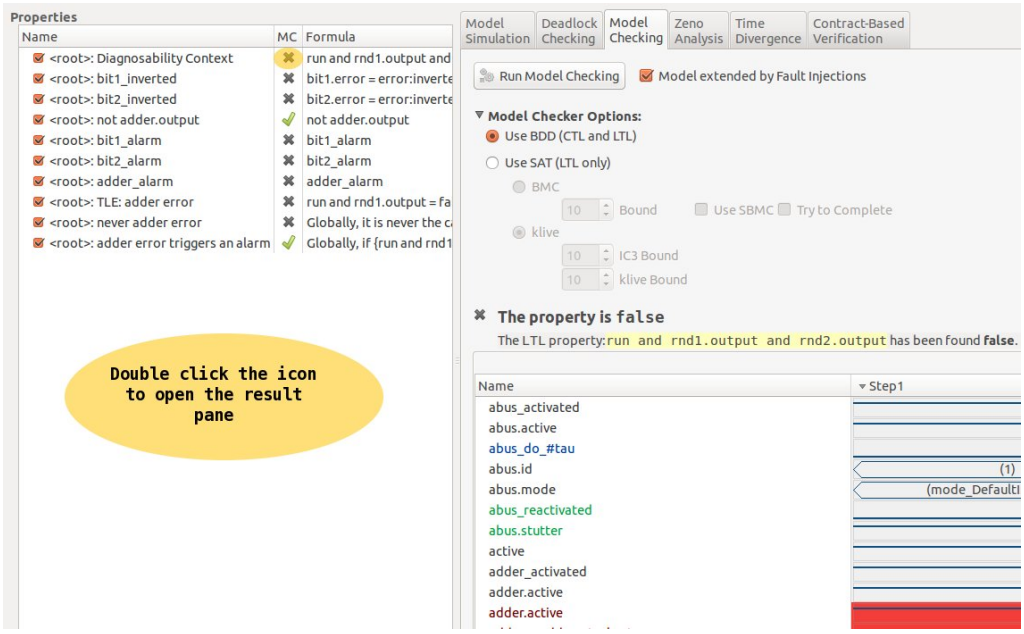


Figure 9.39: Results of model-checking a false property

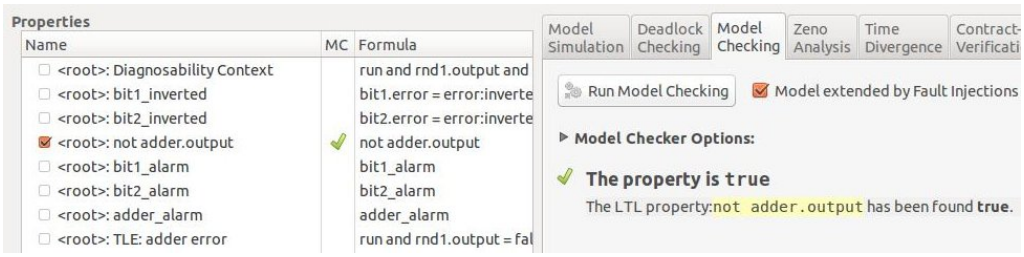


Figure 9.40: The model checking succeeded

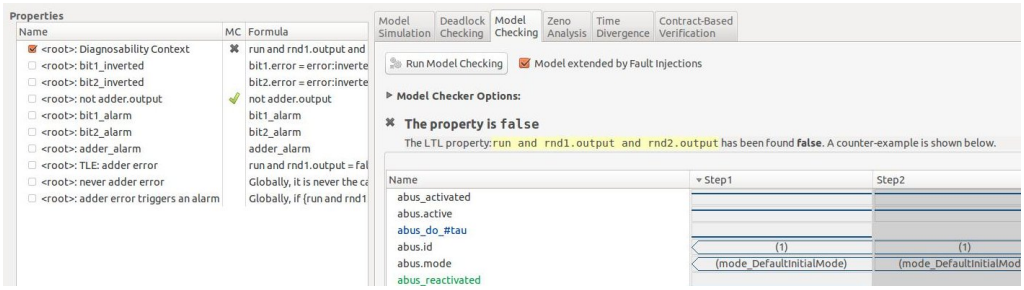


Figure 9.41: A counterexample generated by model checking

1. if a property is satisfied an affirmative answer is shown (compare Figure 9.40),
2. if a property is not satisfied, a counterexample trace is shown (compare Figure 9.41).

When a counterexample trace is produced, the trace can be inspected aimed by filters and all the features already presented in the Section 9.4.2.

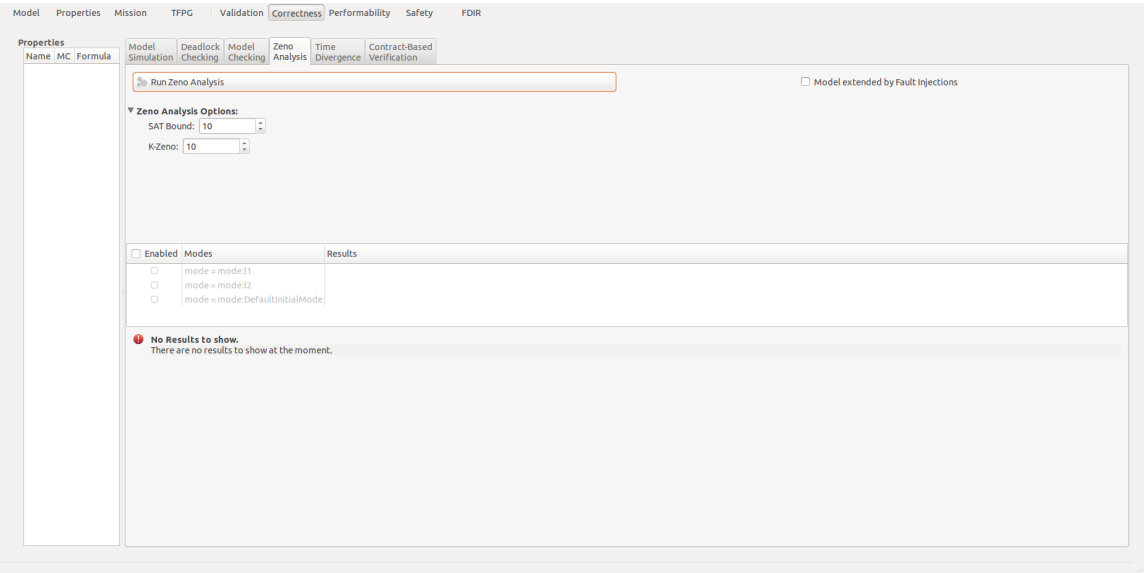


Figure 9.42: Zeno analysis with expanded options

9.4.5 Zeno Analysis

The *Zeno Analysis* pane, see Figure 9.42, is used to check if a mode of the SLIM model is Zeno, or if it contains a timeclock, as explained in more detail below. A mode in a timed/hybrid SLIM model is Zeno if all the finite paths that reach the mode cannot be extended of an infinite non-Zeno path (a path where time diverges). Zeno modes should be avoided in the model, since they represent modes of the system where time cannot diverge. This situation can be due either to a timelock (i.e. a mode where time cannot diverge) or to cycles where time cannot diverge (intuitively, these cycles represent unrealistic paths of the system that perfrom an infinite sequence of computation steps in a finite amount of time).

The underlying model checker allows for Zeno analysis of infinite state (hybrid/time) SLIM models. In this case the SAT and K-Zeno bounds and model extended by fault injections options are available.

After the Zeno Analysis options have been selected, it is possible to run the Zeno analysis. One or more Zeno modes must be selected in the box pane. Clicking the button *Run Zeno Analysis* will sequentially invoke the model checker for each selected Zeno mode, and the result of each Zeno modes is shown at the end of analysis. There are five possible outcomes in the result pane:

1. if the result for a mode is ZENO, a counterexample trace that reaches the mode is shown
2. if the result for a mode is NON ZENO, an infinite counterexample trace that reaches the mode is shown
3. if the result for a mode is UNREACHABLE, the mode is not reachable, not even with a finite path, and no counterexample trace is shown
4. if the result for a mode is NOT PROVED, the check searched for Zeno paths up to the selected K-Zeno bound. In this case, the check did not find any counterexample trace

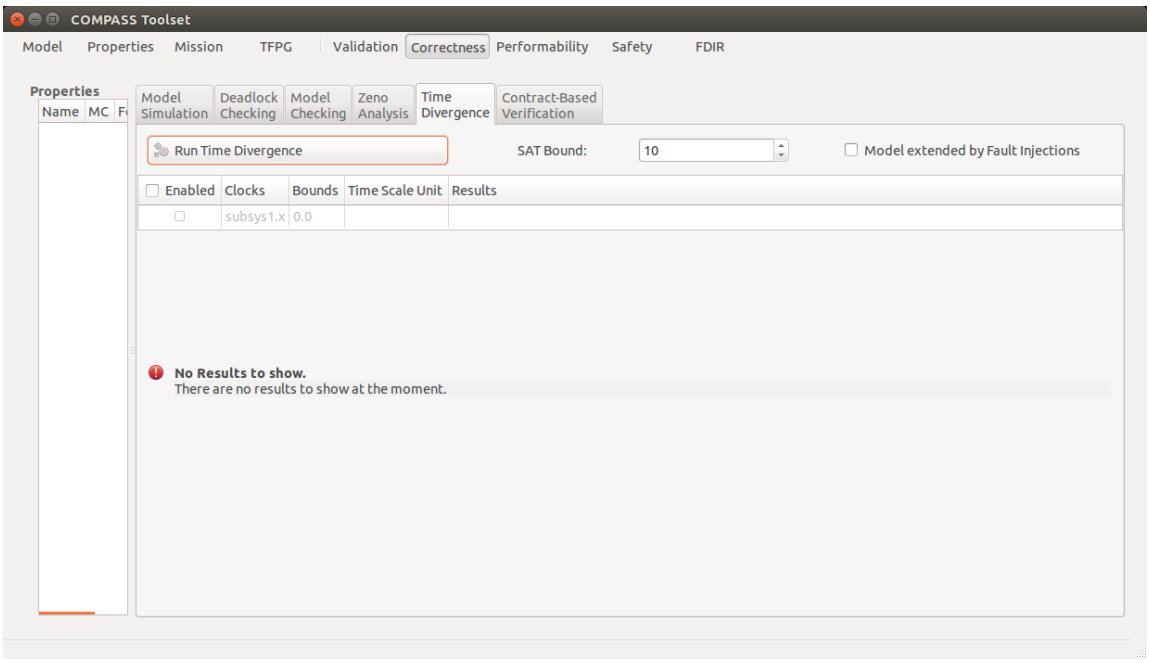


Figure 9.43: Time Divergence

5. if the result for a mode is NOT FOUND, the check cannot find a Zeno-path up to the SAT Bound selected and no counterexample trace is shown

9.4.6 Time Divergence Analysis

The *Time Divergence* pane, see Figure 9.43 is used to detect time divergent clocks in the model. A clock diverges if there exists a trace where its value grows arbitrarily. The implemented time divergence analysis asks the user for the maximum bound of the clock, and checks if there is a path of the system where the clock value is greater than the provided bound.

After the time divergence options have been selected (*SAT Bound*), it is possible to run the time divergence analysis. One or more clocks must be selected in the box pane and it is possible to specify the bound and time scale unit. Clicking the button *Run Time Divergence Analysis* will sequentially invoke the model checker for each selected clock, and the result for each clock is shown when the analysis is completed. There are three possible outcomes in the result pane:

1. if the result found for a clock and a bound is UNBOUNDED the tool shows a finite counterexample trace that reaches a state where the value of the clock is greater than the selected *Clock Bound* is shown.

2. if the result found for a clock and a bound is BOUNDED, then the tool proved that the clock value is always lower or equal than the selected *Clock Bound*, and no counterexample trace is shown.

3. if the result found for a clock and a bound is UNKNOWN, the tool proved that the value of the clock cannot be greater than the specified *Clock Bound* in all the paths of the system shorter thatn the selected SAT bound. However, the check did not prove that

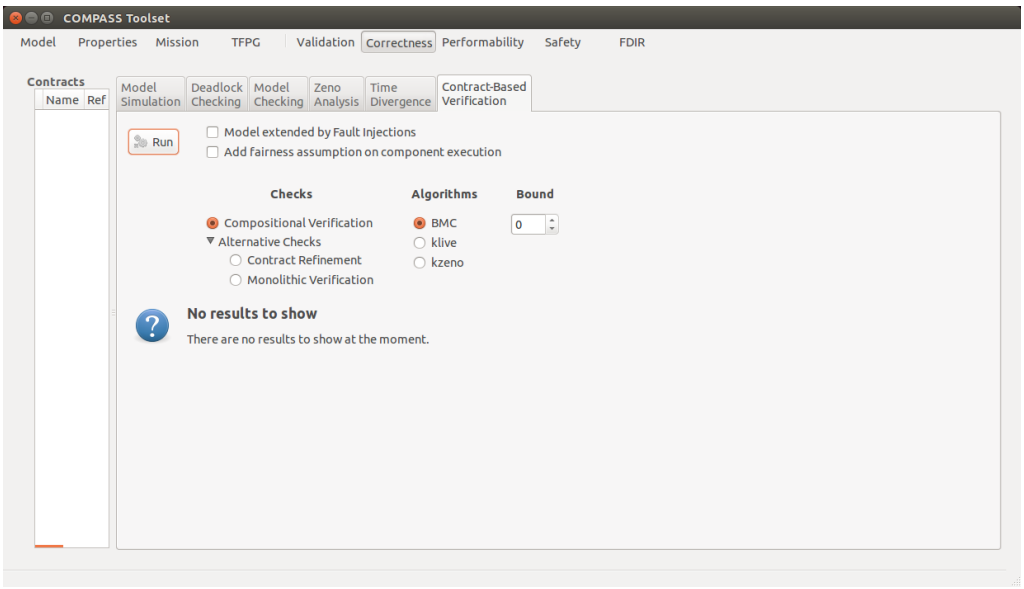


Figure 9.44: Correctness

the clock value is bounded for all the possible system's traces. Also in this case there are no counterexample traces.

9.4.7 Contract-based Verification

This panel allows the same contract refinement validation to be performed as described in Section 9.2.2, with the addition of Monolithic Verification. The *Checks* configuration options allows the verification to be performed to be selected. By default, *Compositional Verification* is selected and performs both contract refinement analysis, as well as monolithic verification. When *Alternative Checks* is expended, both analyses can be selected individually. See also Figure 9.44.

Steps

1. Select the SLIM model from the *Model* pane
2. On the left hand side, select a contract from the *Contracts* pane; this enables the button *Run*
3. Select the *Check* and the *Algorithm* for the analysis
4. The user can now run the analysis by clicking on the button *Run*;

In figure 9.45 Contract-based Verification analysis has been run on the contract `termination` selecting the *Compositional* check; all components results to be ok.

9.5 Performability Analysis

Performability analysis is used to quantitatively analyze a SLIM model from a probabilistic perspective. It requires that fault injections are specified over the SLIM model with proba-

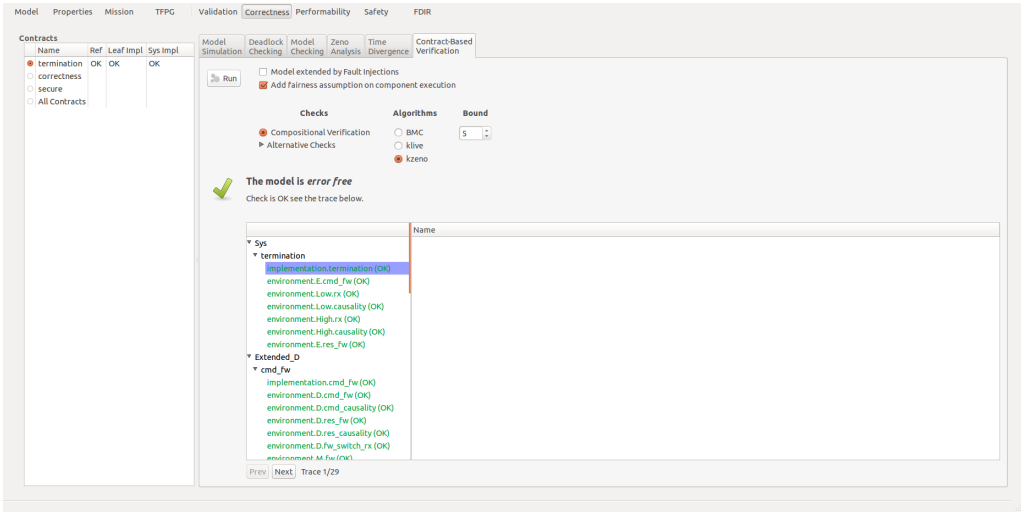


Figure 9.45: Contract Based Verification - Compositional Check

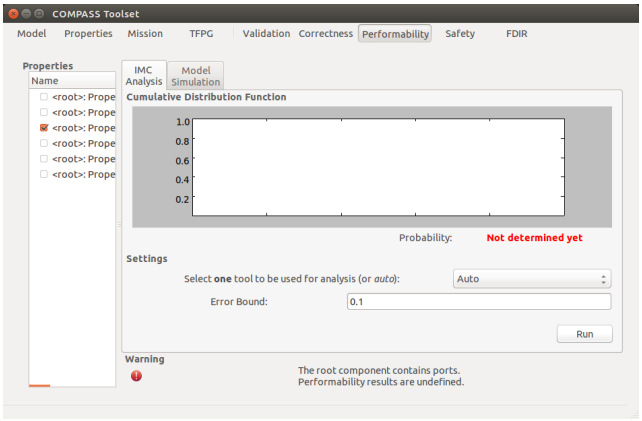


Figure 9.46: Main Performability view

bilistic events.

Steps

1. Load the SLIM model (see Section 6.1).
2. Specify fault injections (see Section 6.3).
3. Specify probabilistic properties (see Section 7).
4. Click on *Performability*.
5. Check one of the probabilistic properties.
6. (Optional) Configure the tool options.
7. Click *Run*. This runs performability analysis.

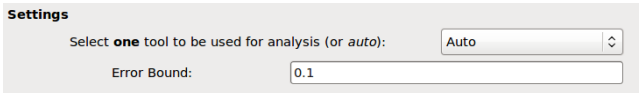


Figure 9.47: Performability Options

Only one probabilistic property can be analyzed at a time. To stop an analysis, click the *Stop* button. The performability tool parameters can be configured as shown in Figure 9.47. The tool to be used can be selected, and the error bound (for the IMCA tool) can be specified.

When performability analysis has finished for a reachability property, a graph appears (see Figure 9.48. The graph describes the *cumulative distribution function* for the chosen probabilistic property. The y-axis is the probability. The x-axis is the upper time bound. A point (x, y) on the graph means that the chosen probabilistic property holds with probability y from the lower time bound (specified in the property) up to the time x. For this reason, the graph always stays constant or is increasing, as the at the next upper time bound, the probability is always equal or higher. The maximum probability, indicated in red, denotes the probability that the property holds from the lower time bound up to the upper time bound (both specified in the property).

Note that if the propositions in the probabilistic property already hold from the initial state and the lower time bound is 0, the probability is always 100% over the complete duration. This follows from the fact that the initial state starts at time 0.

Due to non-determinism, it is possible that two functions are shown in the graph, indicating the minimum and maximum probabilities. This indicates that the probability of the properties lies somewhere between these two functions.

For expected time properties, only the value of the property will be displayed, indicating the time that is expected to be required to reach the state in which the property is satisfied. Similarly, for long-run average properties, only the percentage is displayed that indicates the average time that is spend in states that satisfy the property.

Note that for long-run averages values only make sense if they are specified for states that contain a probabilistic transition. This is due to the "maximal progress assumption", that specifies that the residence time of a non-probabilistic state is zero. Therefore, the average waiting time in such states is 0%. Furthermore, non-zero values can only occur for states that occur infinitely often. This is due to the fact that the long-run average is measured over an infinite amount of time, and as such states that occur only finitely often do not add to the overall waiting time.

Tool settings

Performability analysis can be performed using two different tools: MRMC and IMCA. By default, the toolset will automatically detect and select the tool that best matches the given inputs. It is possible to override this by selecting the corresponding tool from the drop-down list. However, neither tool supports the full spectrum of performability analyses, and if analysis is not possible with the currently selected tool a message will be displayed (see Figure 9.49). The MRMC tool is capable of analyzing deterministic models only for all probabilistic reachability properties. The IMCA tool can analyse non-deterministic models for all probabilistic reachability properties, as well as long-run average and expected time properties. However, for reachability properties that specify an time interval starting from a non-zero value, the

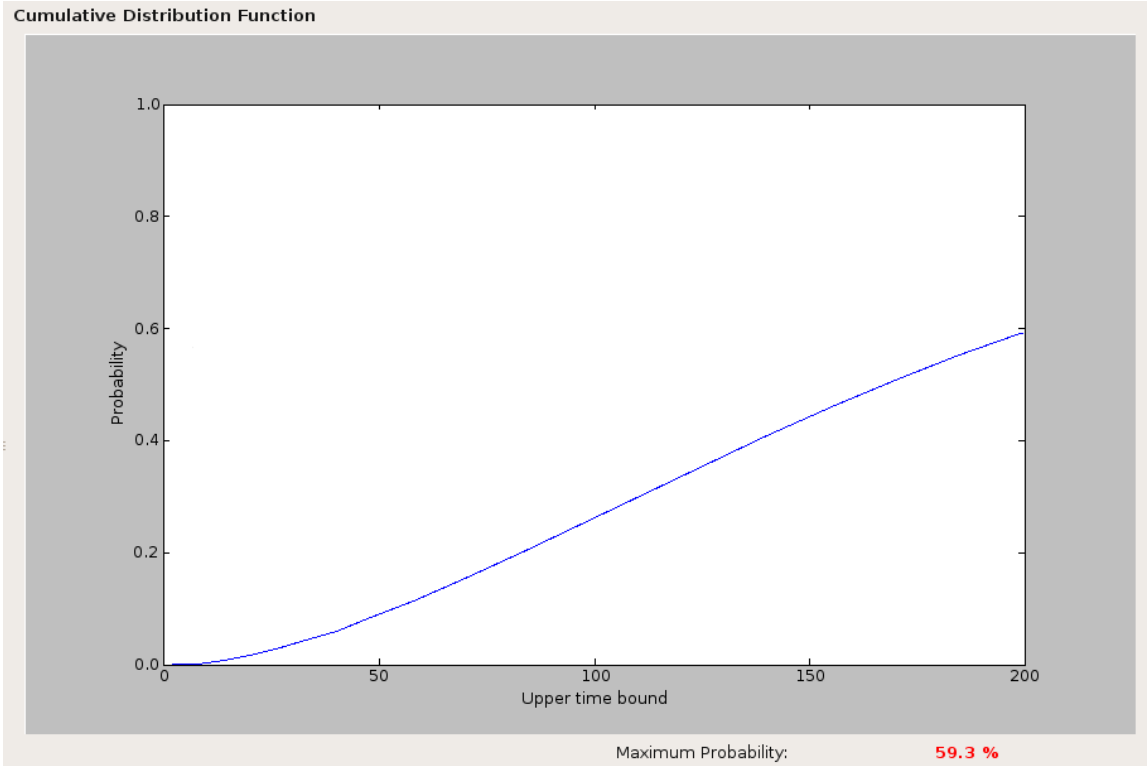


Figure 9.48: Result of Performability Analysis

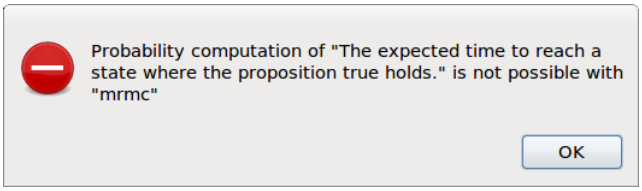


Figure 9.49: Result of Performability Analysis using an incompatible tool

graph is not available, and only the final reachability property is displayed.

Deadlock Avoidance

For all analyses, deadlock checking is recommended to ensure that the model is globally deadlock free (see Section 9.4.3). To ensure that your models are deadlock free, we strongly discourage you to add unguarded self-loops in the nominal model. This aggressively reduces the ability to do performability analysis. The semantical reason for this is the "maximal progress assumption". An unguarded self-loop is always enabled, and as self-loops in the nominal model always have preference over probabilistic transitions in the error model, the latter are cut away. Our recommendation is to avoid deadlocks by solving the true underlying problem of the deadlock.

Non-Determinism

Non-determinism has two sources, the probabilistic property and the model itself. To avoid non-determinism we recommend you to:

- Only specify probabilistic properties over error states (like "`sensors.sensor1.error = error:Dead`"). Non-determinism is more likely if the property also reasons over values of data elements or modes in component implementations.
- Ensure that all error events in the error model have a poisson rate associated. Non-determinism is more likely if probabilistic information is missing.

9.5.1 Relation to Fault Tree Generation

Both performability analysis and Fault Tree Generation (see Section 9.6.1) are based on the state space of the system. The difference lies in the abstraction. Whereas fault tree generation abstracts the state space into a tree shape, performability analysis considers a more fine-grained abstraction of the state space using a notion of weak bisimulation equivalence. Both abstractions have their drawbacks. Fault tree generation generates at most PAND-gates for capturing orderings of events. Performability takes more subtle orderings in account, like those induced by repairs. The drawback is that this increased range of expressiveness limits the supportable systems to deterministic and deadlocking systems. The theory to practically support non-determinism and non-deadlocking systems is future work.

The degree of abstraction of fault tree evaluation versus performability has an impact on the computed probabilities. As performability abstracts the state space in a more fine-grained manner than fault tree evaluation, its computed probabilities reflect the input system better.

9.5.2 Choice of Duration Parameter

The unit of time used by the time bounds in the probabilistic properties and the one in which the rates are expressed must be semantically of the same order. Thus if for example the rates are expressed per hour, the time bounds must also be expressed per hour. The granularity of the chosen time unit has an effect on the numerical performance. We recommend the coarsest time unit possible without losing numerical significance. This leads to the fastest performance and the highest achievable numerical stability.

9.5.3 Choice of Error bound Parameter for IMCA

The error bound indicates to what accuracy the results are computed. That has a direct influence of the number of steps, and therefore the time, required to compute the results. The number of steps is determined by three factors: The error bound (inversely quadratic), the maximal exponential rate (quadratic) in the model and the upper time bound of the property (proportional). By lowering the error bound, more accurate results can be obtained, and the cost of calculation time. An error bound of 0.01 indicates that the results is accurate up to 1%.

In order to speed up modeling, a higher error bound may be specified initially to get faster results. This allows for quick iteration of the model or property specifications. Afterwards, a lower error bound can be used to get more fine grained results.

9.5.4 Numerical Stability for MRMC

Numerical stability is composed of three intertwining factors: the approximation error, the rounding errors and the errors due to overflowing and underflowing.

The approximation error is due to the used numerical algorithm. The algorithm is essentially a finite approximation of the infinite expression that represents the probabilities. The remaining infinite terms that are not accounted for in the finite approximation is the approximation error. The Krylov algorithm works as an iterative algorithm that expands the size of the finite expression until a predefined error bound is met. The current error bound is set to 10^{-6}, meaning that the approximation is analytically correct up to 6 digits behind the point.

The rounding errors occur during most arithmetic operations and depend on the input and the arithmetic operation itself. As numbers are represented using a finite amount of bits, arithmetic operations over them can lead to new values that cannot be represented sufficiently with the machine word size (32-bits or 64-bits, depending on the used machine). Rounding errors are not accounted for in the error bound of 10^{-6}. Overflowing and underflowing occur when the computed (intermediate) results are too small or too large to be represented using the machine word size. Depending on the machine architecture, a NaN or 0 becomes the result instead. Both overflows and underflows are not checked by the current implementation.

Since the computed probabilities are influenced by these three factors, of which only the approximation error is controllable, we provide a table of benchmarked models (cf. Table 9.2) for which we validated that the computed probabilities are numerically correct despite the presence of the three sources of possible instabilities. Note that the states and transitions metrics are derived from the underlying Markov chain that is computed from the system's state space, and that the Markov chain is strictly smaller than the latter. The user can check the Markov chain's metrics by opening the .tra file in the debugging folder. The stiffness is the ratio between the smallest and largest rates and are derived from the occurrence rates in the error models. This table must be considered as a guideline.

Model	States	Transitions	Stiffness
CSPS	3072	14848	1600
TQN	861	2859	400
PTP	1024	5121	0.5
ER	4011	11431	4000000
WGC	1329669	9624713	6164

Table 9.2: Model properties of the case studies.

9.5.5 Simulation

In case performability analysis by means of Markov models is not available, Simulation can be used instead. Simulation is performed by means of Monte Carlo analysis, which uses a statistical approach to generate the resulting performability metrics. The main simulation view contains some settings to control the accuracy and type of simulation, see Figure 9.50. The error bound controls the size of the error margin in the resulting probability, where a bound of 0.05 means the result lies within $+/-5\%$. The confidence determines the likelihood

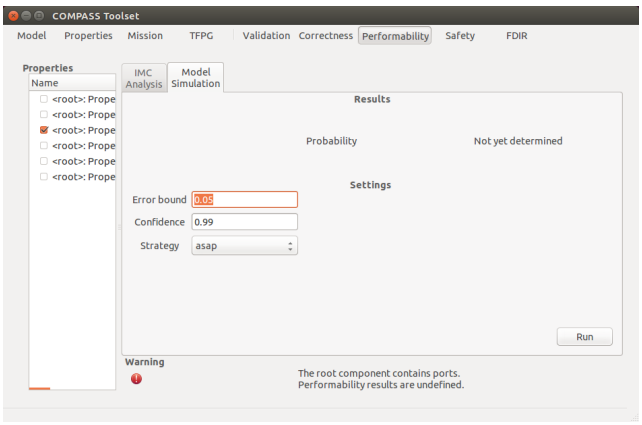


Figure 9.50: Simulation Performability view

the resulting probability matches the actual probability. Due to the fact that a simulation is used, the result cannot be guaranteed with a confidence of 100%, but can be determines arbitrarily close to it. Finally, the strategy determines how non-deterministic time intervals are treated.

Not all probabilistic properties are supported. Simulation is limited to only those properties that do not contain nested probabilistic operators. When using the property patterns, that means probabilistic response patterns are not supported. Furthermore, the expected time and long-run average metrics are not supported either.

Non-determinism

Due to the nature of the simulation semantics, the model must be assumed to be fully probabilistic, meaning non-determinism is not allowed. In case non-determinism is present in the model, it is resolved in the following ways:

- Non-deterministic choices of transitions are resolved in an equi-probabilistic manner, i.e., all choices are assigned the same probability.
- Non-deterministic time delays are resolved according to the configured strategy:
  - ASAP: Take a transition as soon as it is available;
  - Local: Take a transition at a random time as determined by its local invariant;
  - Maxtime: Delay all transitions as much as possible;
  - Progressive: Like Local, but permits delaying up to a time point at which point a transition is not available.

9.6 Safety and Dependability Analysis

The *Safety* window of the toolset 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. It is composed of six sub-panes:

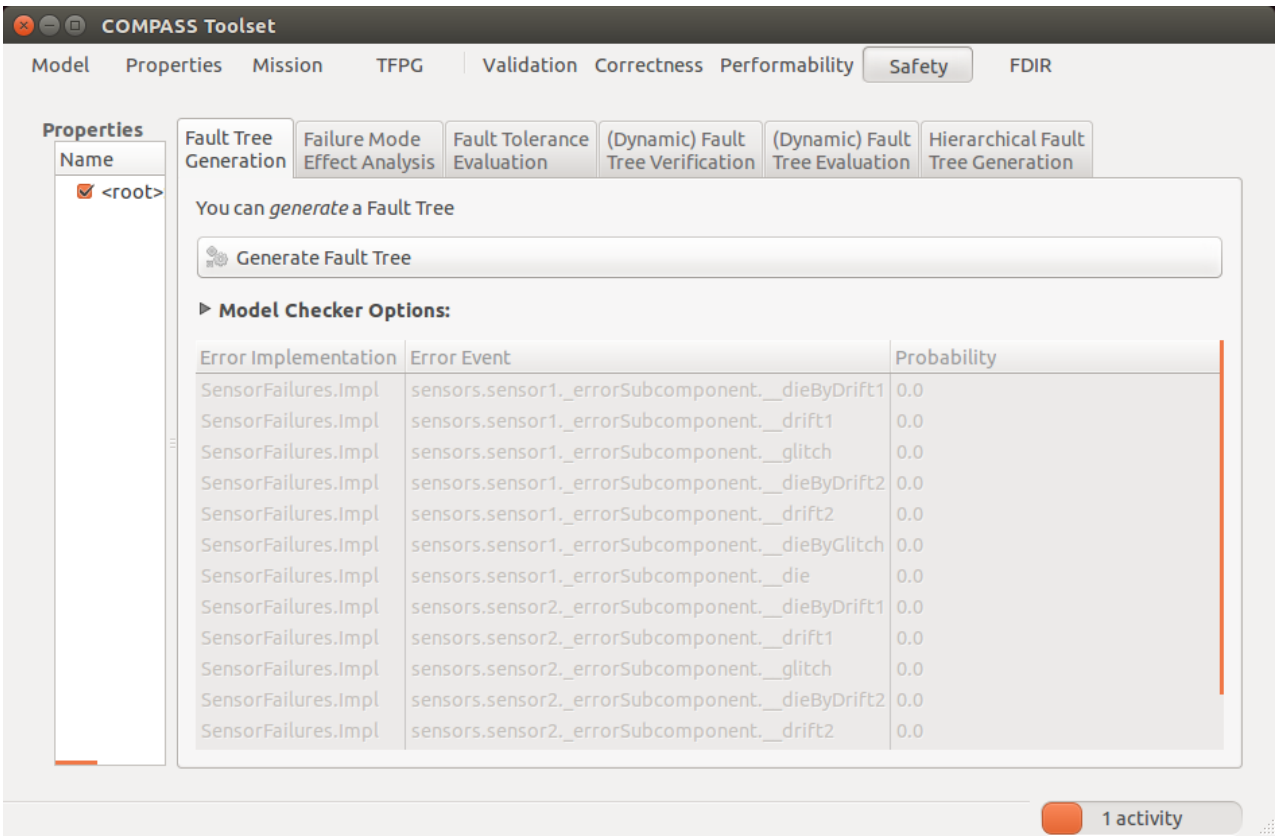


Figure 9.51: The main *Safety* pane

1. *Fault Tree Generation* allows the user to generate a fault tree that shows the minimal sequence of events that may lead to a failure state
2. *Failure Mode and Effects Analysis* shows tabulates all the possible combinations of events that may lead to one or more failure states
3. *Fault Tolerance Evaluation* computes a measure of fault tolerance, given a set of already generated fault trees, based on the cardinality of their cut sets
4. *(Dynamic) Fault Tree Verification* plots the probability graph given a (dynamic) fault tree and a probabilistic property specified over it
5. *(Dynamic) Fault Tree Evaluation* computes the probability of the top-level event to occur along with the criticality
6. *Hierchical Fault Tree Generation* allows user to generate a fault tree based on contracts

The image in Figure 9.51 shows a screenshot of the window before any interaction with the user: The *Properties* list should not be empty, in order to run the analyses described in this section.

9.6.1 Fault Tree Generation

The *Fault Tree Generation* pane shows for a given property (created with the properties pane) which are the minimal combinations of events that are possible explanations for the the failure

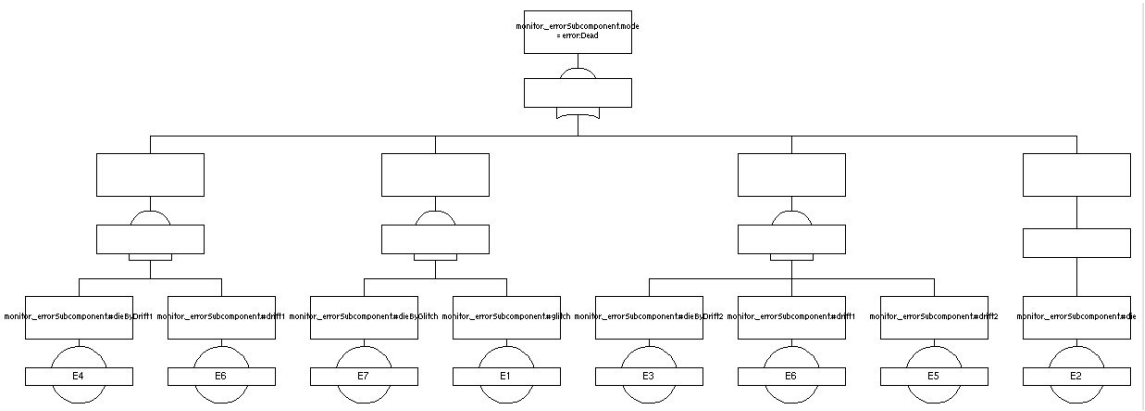


Figure 9.52: An example of fault tree

specified in the property.

Three possible engines are available:

1. IC3 (the defalt)
2. BDD (if the model is not hybrid)
3. SAT

Moreover, the *Dynamic* option is not supported for hybrid models (compare section 9.1). For more information on the 'sat bound' option see Section 9.4.4.

Steps

1. Select the SLIM model from the *model* pane.
2. Specify one or more fault injections.
3. Specify a property to check in the *properties* pane.
4. In order to generate a fault tree for a specified property, the user has first to select the property. As a consequence, the *Generate Fault Tree* button becomes enabled.
5. After clicking on the *Generate Fault Tree* button, a popup appears with a fault tree displayer inside. The fault tree shows how it is possible to reach the state corresponding to the property, i.e. what are the sequences of events that may cause the failure, linked by 'and' and 'or' gates (compare Figure 9.52). In this example there are:
   (a) one branch (called cut set) with a single event that causes the fault,
   (b) two branches with a sequence of two events that together cause the fault (linked by an 'and' gate), and
   (c) one branch with a sequence of three events that together cause the fault (linked by an 'and' gate).

The fault tree generated will be stored, in order to be used for other analysis, (e.g. for Fault Tree Evaluation, see Section 9.6.5) that require a property associated with a fault tree in order to work.

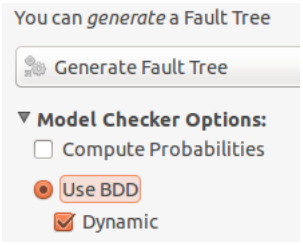


Figure 9.53: Activating dynamic fault tree generation

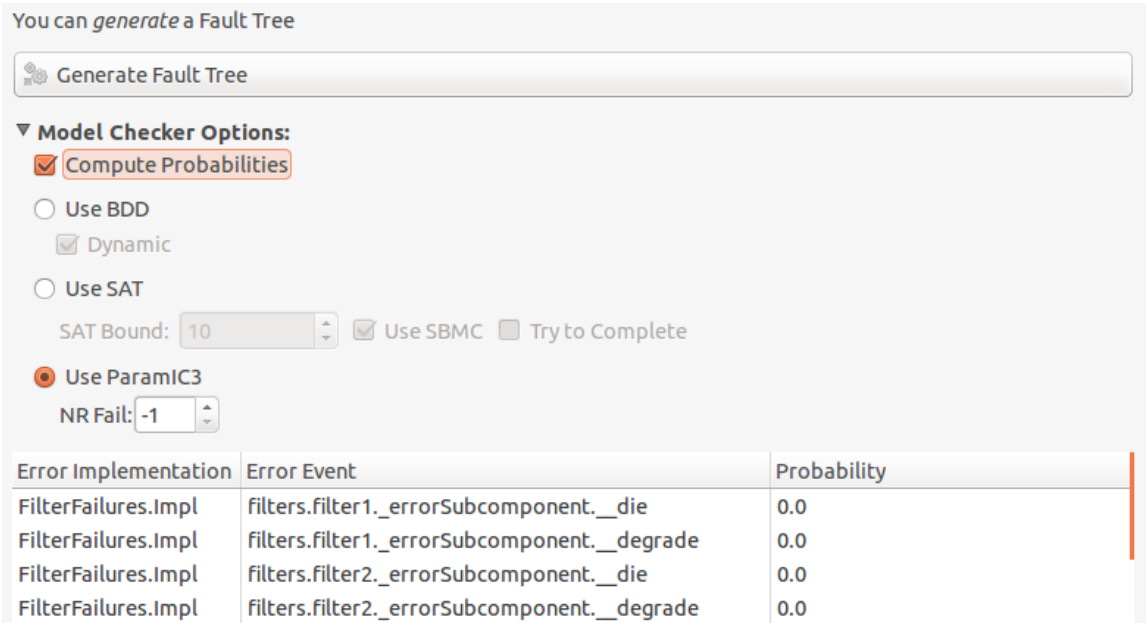


Figure 9.54: Activating probabilistic fault tree generation

9.6.2 Dynamic Fault Tree Generation

The dynamic fault tree generation can be done using the same views shown in Section 9.6.1, with the checkbox *Dynamic* activated, as shown in Figure 9.53. The difference between the dynamic fault tree generation and the fault tree generation is that the "dynamic" one shows also the precedence of the events (i.e., an event must hold before another one in order to reach the failure) using the "priority and", in addition to the "and", gate in the fault tree. Note that the dynamic fault tree generation is disabled when the model is hybrid (see Section 9.1).

9.6.3 Probabilistic Fault Tree Generation

The probabilistic fault tree generation can be done using the same views shown in Section 9.6.1, with the checkbox *Compute Probabilities* activated, as shown in Figure 9.54.

In this case, the box containing probabilities for each fault becomes active and can be edited by the user; the generated fault tree will show also the probabilities of the different nodes (see for instance figure 9.55).

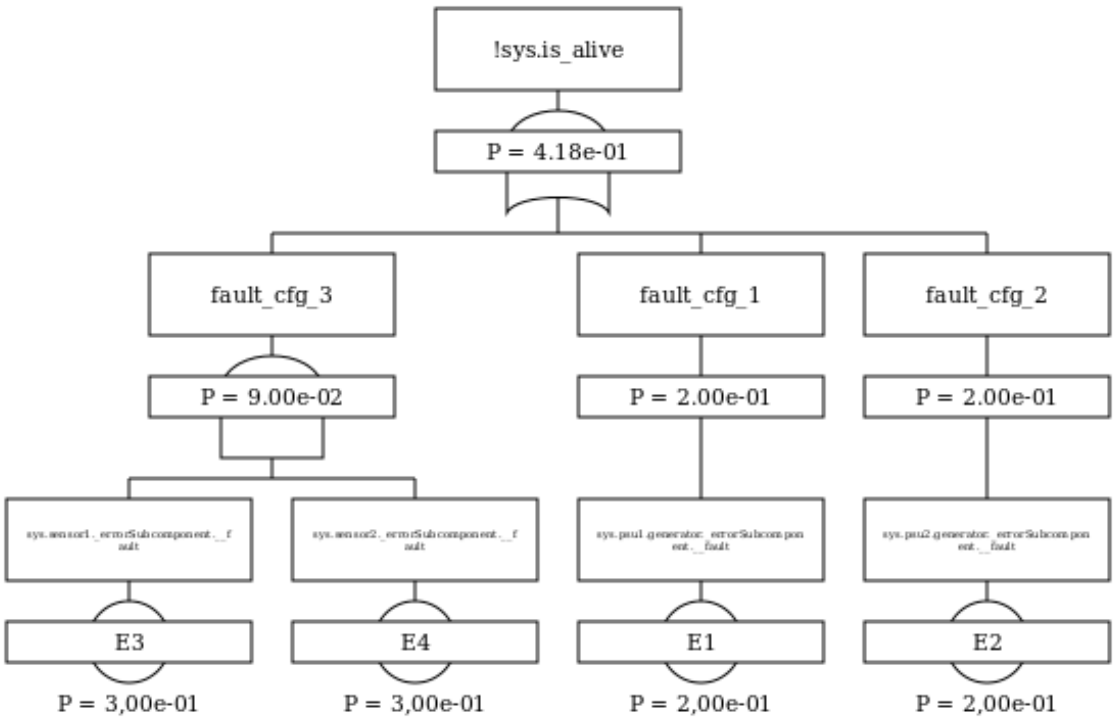


Figure 9.55: Example of probabilistic Fault Tree

9.6.4 Failure Modes and Effect Analysis

This pane allows the user to generate, in tabular form, the set of combinations of events that may cause a given failure. If the checkbox *dynamic* is activated, the order of the events is important, otherwise, it is not.

In Figure 9.56 the main view of this analysis pane is shown.

Steps

1. Select the SLIM model from the *Model* pane.
2. Specify one or more fault injections.
3. Specify one or more property to check in the *Properties* pane; this will enable the button *Generate FMEA Table* (compare Figure 9.57).
4. The user can now run the analysis by clicking on the button *Generate FMEA Table*, that fills the table below with the results of the computation, as shown in Figure 9.58.

In this example, the results show that in order to cause the event

"`value = 0 and filters.mode = mode:Backup`"

(i.e. filters fails twice) there is

1. three combinations of cardinality two:

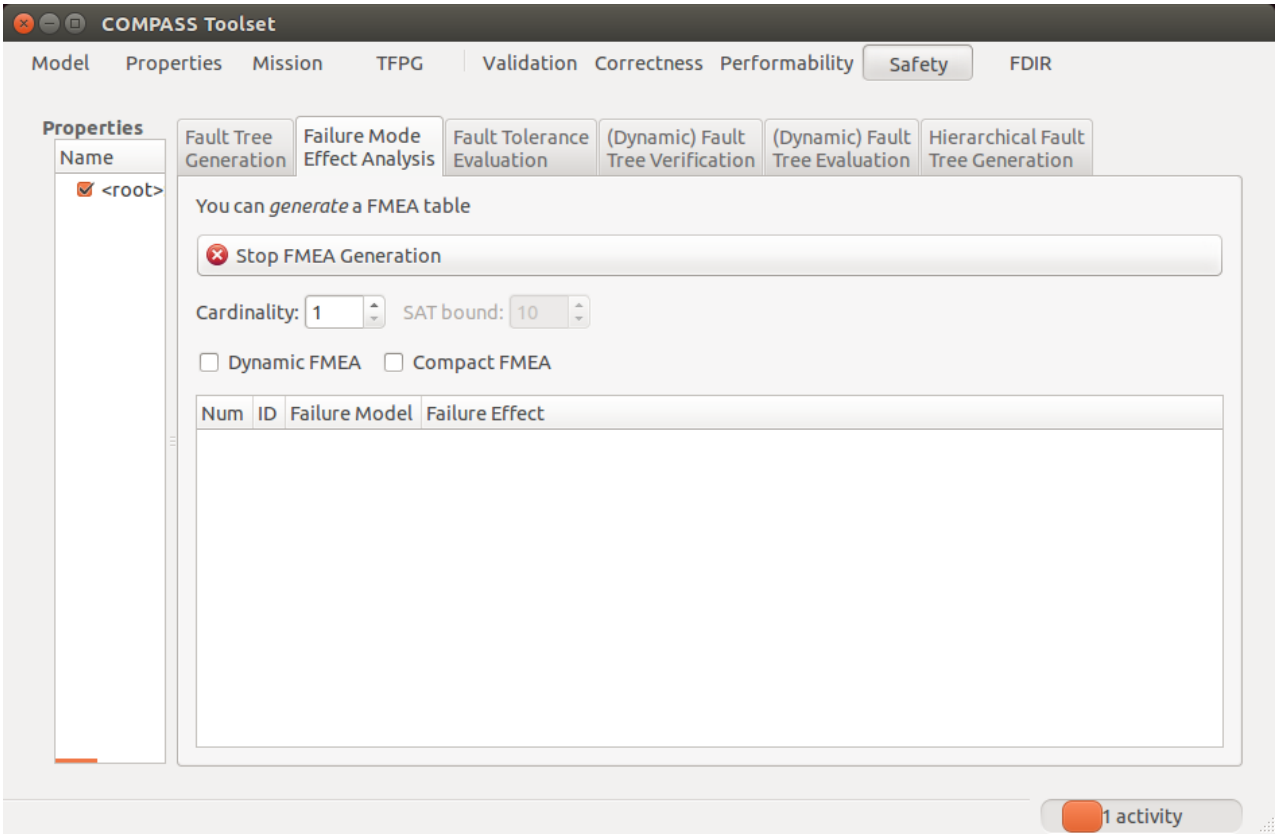


Figure 9.56: The *Failure Modes and Effect Analysis* pane

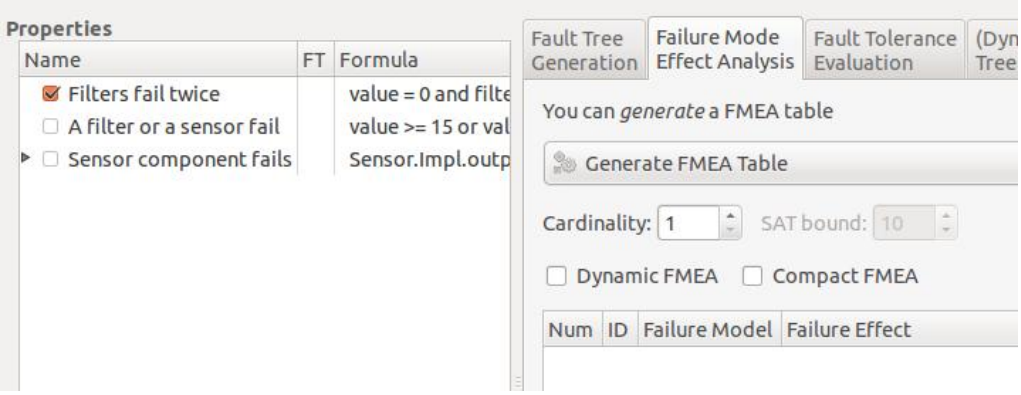


Figure 9.57: The Failure Modes and Effect Analysis enabled

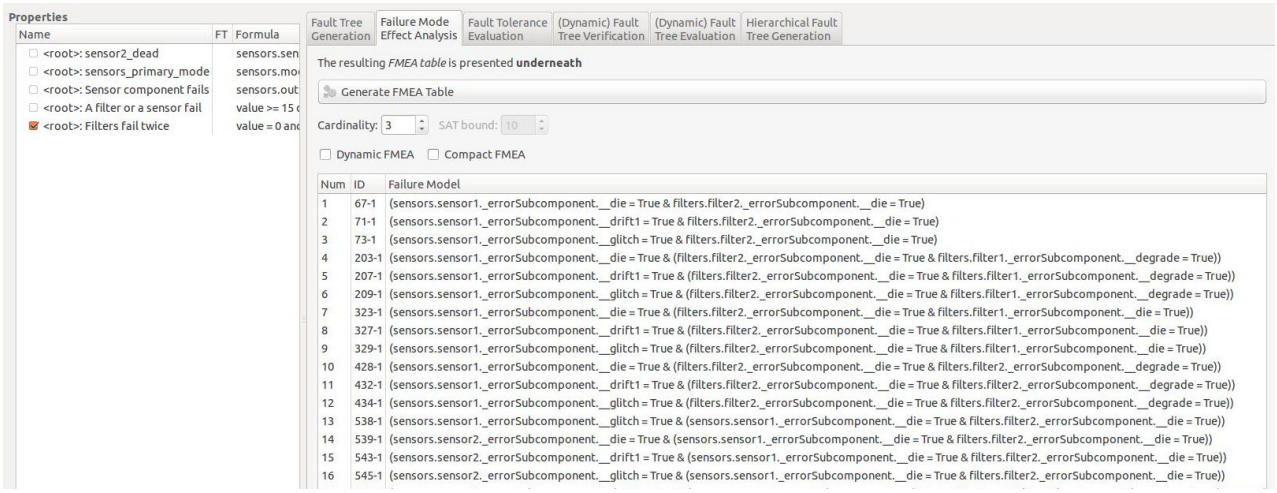


Figure 9.58: The Failure Modes and Effect Analysis results

Figure 9.59: The *Compact FMEA* option

(a) `sensors.sensor1._errorSubcomponent.__die = True & filters.filter2._errorSubcomponent.__die = True`

(b) `sensors.sensor1._errorSubcomponent.__drift1 = True & filters.filter2._errorSubcomponent.__die = True`

(c) `sensors.sensor1._errorSubcomponent.__glitch = True & filters.filter2._errorSubcomponent.__die = True`

as shown in the *Failure Model* column,

2. some combinations of cardinality three, which we do not list here for brevity, but that are visible in Figure 9.58.

The highest computed cardinality can be specified in the *cardinality* field as shown in Figure 9.58. In the example this value is set to 3.

If the *dynamic* option is enabled, the order of the events is taken into account (i.e., if an event occurs before another one, then it must occur before the second one), otherwise the order is neglected.

The column *ID* represents a unique identifier has 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).

When option *Dynamic FMEA* is enabled, the compaction is not available, and the values in the *ID* column are *N/A* (Not Available).

Definition of Compact FMEA Tables

Figure 9.59 shows the option for enabling the generation of Compact FMEA tables. The options *Compact FMEA* and *Dynamic FMEA* are mutually exclusive.

The goal of FMEA compaction is to improve the readability of static FMEA tables as generated by the COMPASS toolset, by avoiding computation of entries that are not interesting from an engineering perspective. Intuitively, an entry is not interesting if it is redundant, that is, if the corresponding fault configuration contains faults that have no effect on the top-level event. The objective of compaction is to compute a *reduced* FMEA table that does not contain such entries. This reduction is meaningful for monotonic systems, i.e. systems such that whenever a fault configuration FC may raise an event E, any proper superset of FC (with additional faults) may still cause E.

More formally, the definition of reduced FMEA tables is as follows. Let $\mathcal{F} = \{F_1, \ldots, F_n\}$ be a set of faults and $\mathcal{E} = \{E_1, ..., E_m\}$ a set of events. A *fault configuration* is a subset of faults $FC \subseteq \mathcal{F}$. An FMEA table is a set of pairs (FC_i, E_j), where FC_i is a fault configuration and $E_j \in \mathcal{E}$ is an event.

Given an FMEA table T, we define the corresponding *reduced FMEA table*, denoted T^R, as follows:

$$T^R = \bigcup_{j=1}^{m} T_j^R$$

where T_j^R, called the reduced FMEA table for event E_j, is defined below, by induction on the cardinality of the FMEA table entries. Let N be the cardinality of the FMEA table T^R that we are computing. Then, we define

$$T_j^R = \bigcup_{k=1}^{N} T_j^R(k)$$

where $T_j^R(k)$, for $k = 1, \ldots, N$ is defined inductively as follows, where $T_j(k)$ denotes the set of entries of FMEA table T_j whose fault configuration has cardinality k and whose event is E_j.

(Base Case) $T_j^R(1) = T_j(1)$

(Inductive Case) $(FC, E_j) \in T_j^R(k)$, for $k = 2, \ldots, N$, if and only if:

- $(FC, E_j) \in T_j(k)$
- one of the following two conditions holds:
  - there exists no proper subset $FC' \subset FC$ such that $(FC', E_j) \in T_j^R(h)$ for an integer h such that $h \in \{1, \ldots, k-1\}$
  - there exists a cover of FC, that is, a set of fault configurations FC_i, and a set of integers h_i, with $i = 1, \ldots, l$, such that $FC = \bigcup_{i=1}^{l} FC_i$ and $(FC_i, E_j) \in T_j^R(h_i)$, with $h_i \in \{1, \ldots, k-1\}$

Intuitively, the first case covers the situation in which we have a "genuine" new entry in the FMEA table, which is not subsumed by any entry of lower cardinality. The second case formalizes the notion of (non-)redundancy by means of the notion of "cover". Note that, by definition, a reduced FMEA table T^R is always a (possibly improper) subset of the original FMEA table T.

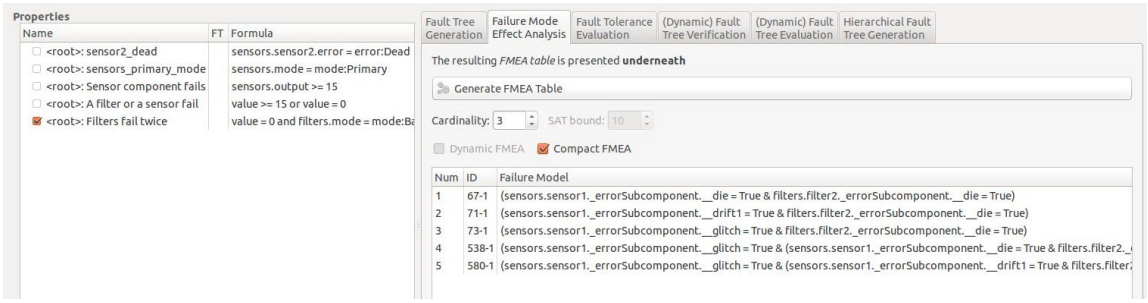


Figure 9.60: The compacted FMEA table result

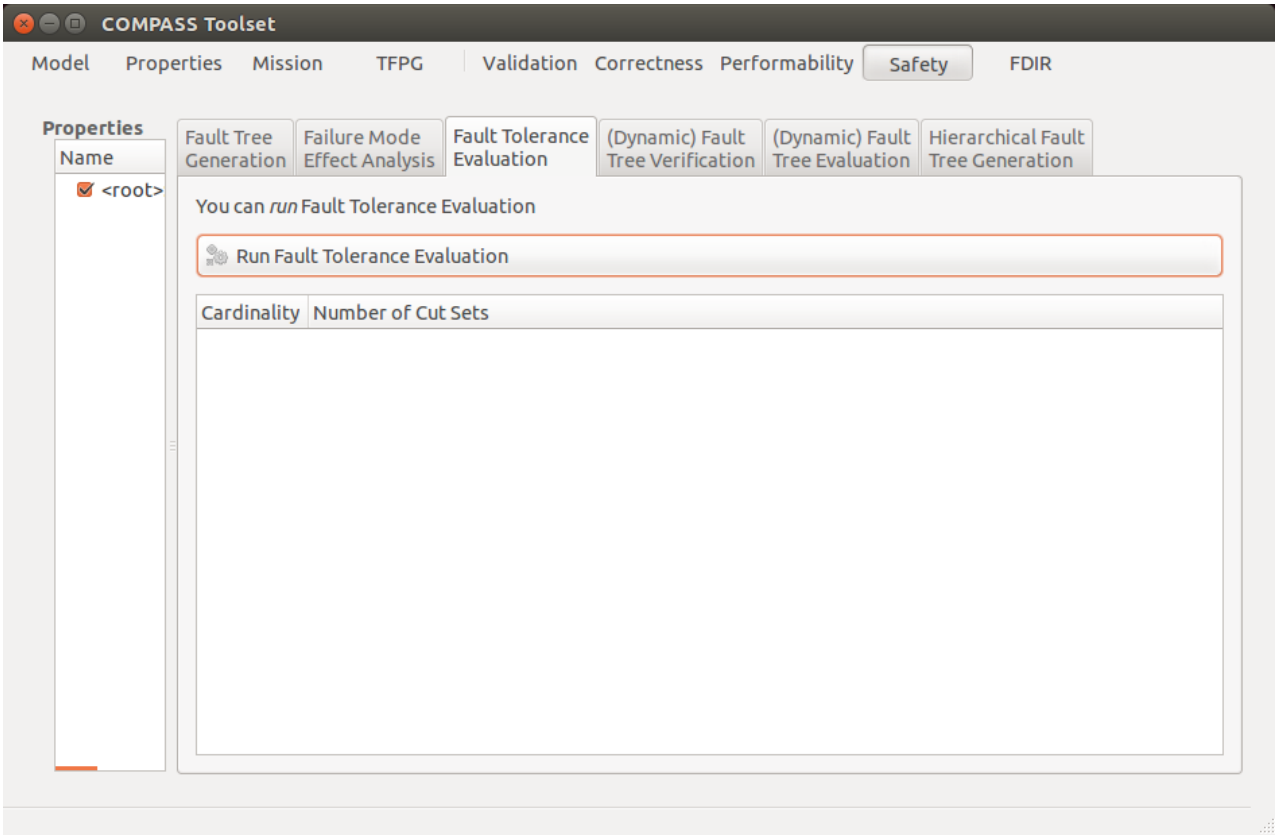


Figure 9.61: The *Fault Tolerance Evaluation* pane

Figure 9.60 shows the result of the same FMEA analysis shown in Figure 9.58, but this time run with the option *Compact FMEA* enabled.

9.6.5 Fault Tolerance Evaluation

The fault tolerance evaluation window allows the user to compute a measure of fault tolerance, given an already generated set of fault trees. Figure 9.61 shows the main view of the tool for this feature.

Steps

1. Select the SLIM model from the *Model* pane.

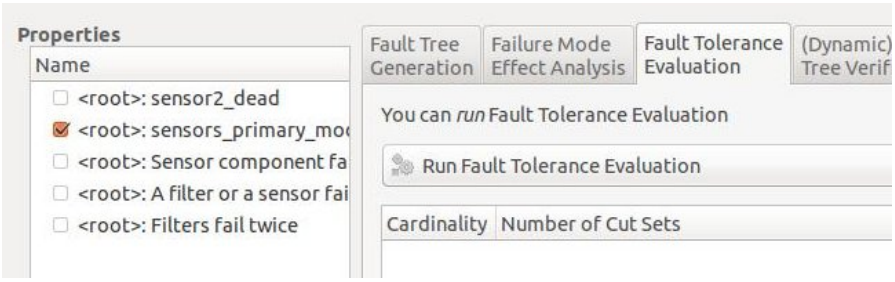


Figure 9.62: The Fault Tolerance Evaluation main button enabled

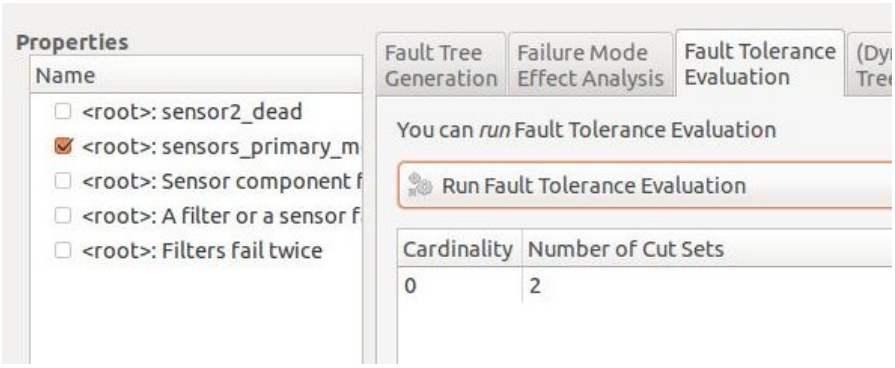


Figure 9.63: The Fault Tolerance Evaluation results

2. Specify one or more fault injections.
3. Specify one or more property to check in the *Properties* pane.
4. In order to enable the button *Run Fault Tolerance Evaluation*, first the user has to select one or more properties that have an associated fault tree. (In order to associate a fault tree to a given property it is sufficient to run fault tree generation as described in Section 9.6.1). An example is shown in Figure 9.62.
5. the user can then click on the button *Run Fault Tolerance Evaluation* that fills the table shown in Figure 9.63

The analysis counts the number of *unique* cut sets computed for all the generated fault trees, divided on the basis of their cardinality. As an example, the results in Figure 9.63 show that there are only two cut sets of cardinality zero. Another example is given in Figure 9.64. Here there are:

1. One cut set with cardinality 1
2. Two cut sets with cardinality 2
3. Two cut sets with cardinality 3

9.6.6 (Dynamic) Fault Tree Evaluation

As evaluation over dynamic fault trees is an extended variant of evaluation over non-dynamic fault trees, the functionality for their evaluation is combined in a pane. It is used to compute

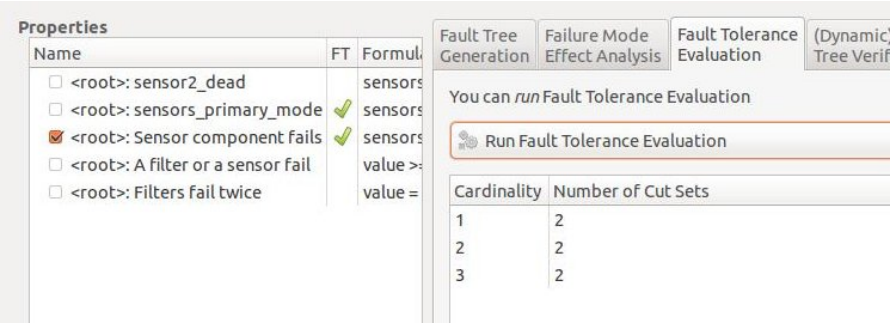


Figure 9.64: Another example of Fault Tolerance Evaluation results

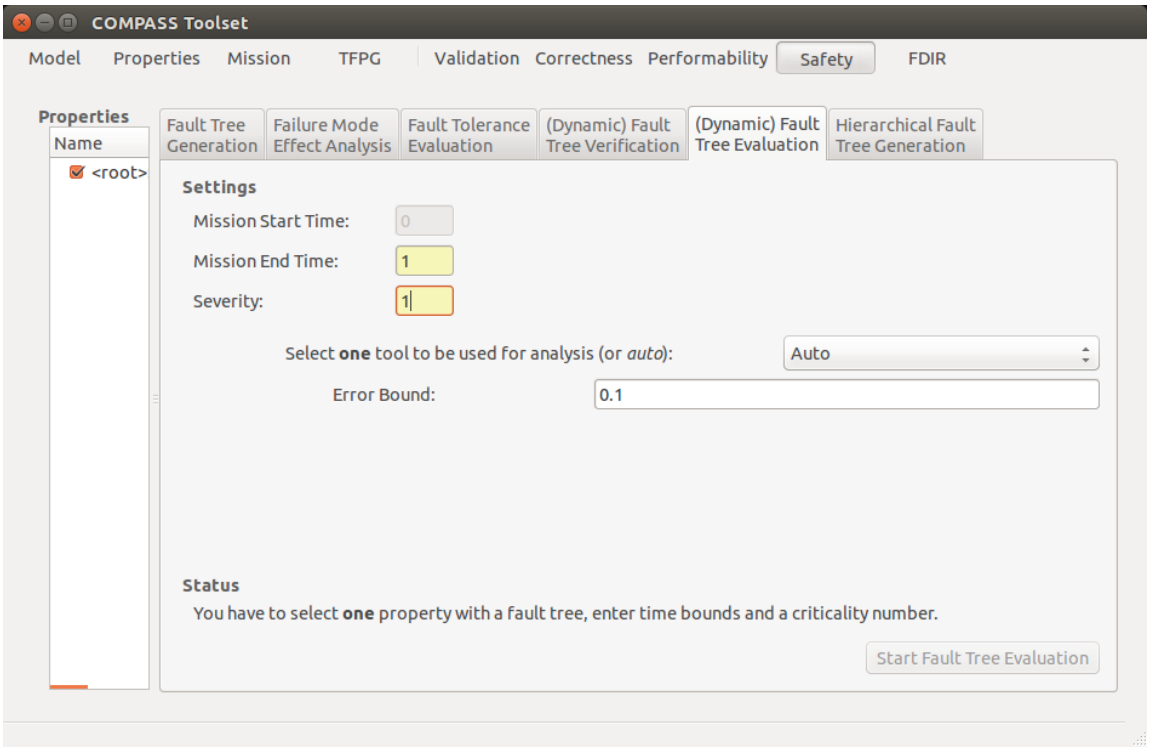


Figure 9.65: Entering the input values for (Dynamic) Fault Tree Evaluation.

the probabilities that the gates and the top-level event are triggered from time point 0 until the upper time bound.

Steps

1. Perform (Dynamic) Fault Tree Generation (see Section 9.6.1).
2. Click on the "(Dynamic) Fault Tree Evaluation" tab (see Figure 9.65).
3. In the properties pane on the left, check a property that has a (dynamic) fault tree associated.
4. Enter the upper time bound. The mission time will be then set from 0 to that bound.

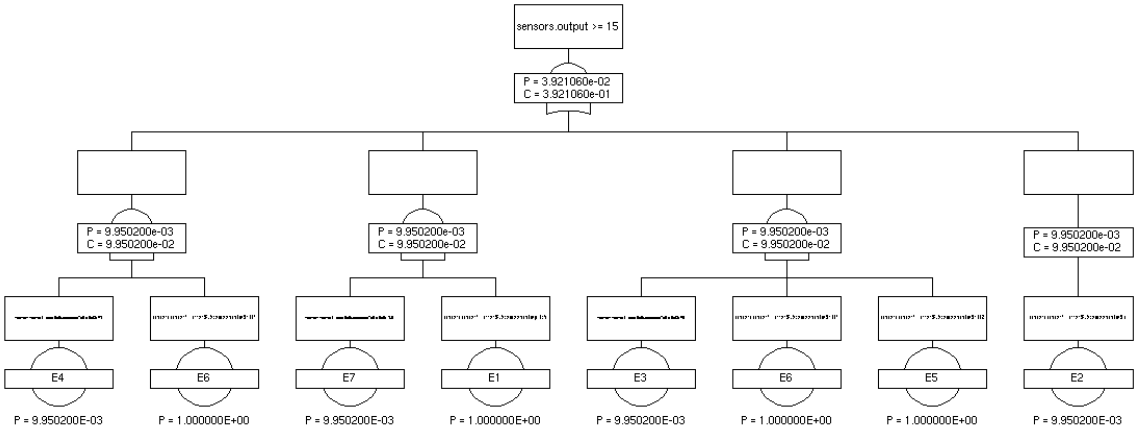


Figure 9.66: Result of (Dynamic) Fault Tree Evaluation.

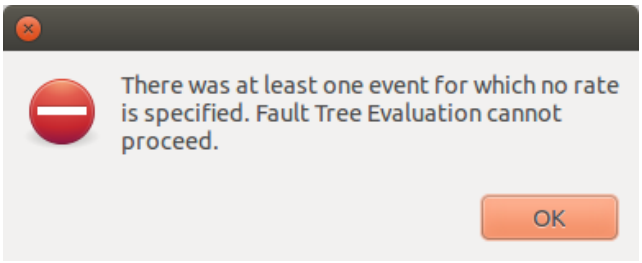


Figure 9.67: Lack of basic events to poisson rates error upon (dynamic) fault tree evaluation.

5. Enter the Criticality for the triggering of the top-level event of the chosen fault tree. This number is used for Criticality Evaluation (see Section 9.6.7)

6. Click on "Start Fault Tree Evaluation"

The result of (dynamic) fault tree evaluation is the same (dynamic) fault tree with probabilities attached to the gates and the top-level event (see Figure 9.66). The probabilities are denoted after the "P = " part of a gate.

Note that two errors could be issued upon (dynamic) fault tree evaluation:

- Not all basic events have a poisson rate associated. An error dialog occurs as shown in Figure 9.67. To prevent this from occurring, we recommend you to ensure that all error events in error model implementations have poisson rates associated with it.

- Non-determinism could occur. The foremost reason for this to happen is that basic-events are shared by different cut sets, and that upon triggering of a shared event, the top-level event could be triggered. The underlying reason is non-determinism in the SLIM model or that the top-level event is underspecified. We recommend you to check whether the purely propositional property truly represents the top-level event.

Evaluation of Dynamic versus Static Trees

Dynamic and static fault trees are computed from the system's state space, and the difference lies in that the former captures in more detail ordering of events due to the addition of more

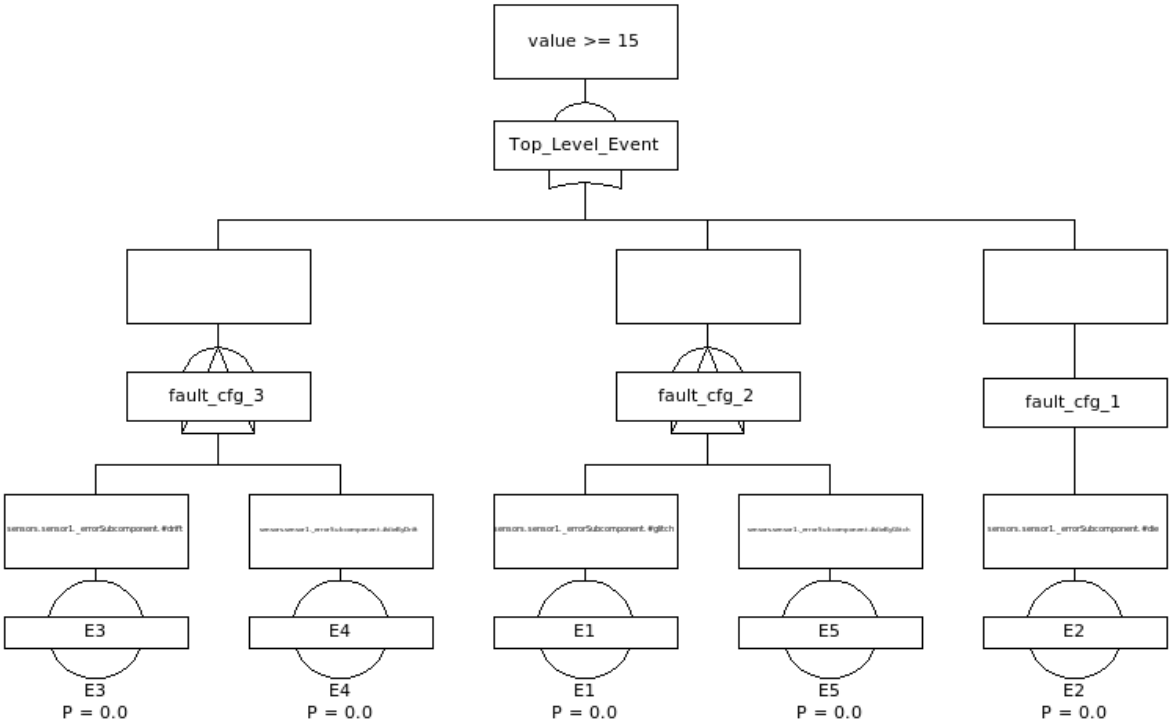


Figure 9.68: Dynamic Fault Tree with basic events E1 to E5. Events E3 and E4 are ordered, and so are E1 and E5.

expressive gates. Even though static fault trees also account for orderings, they are only captured coarsely using the boolean gates. The increased amount of detail in dynamic fault trees may cause that an evaluation leads to more precise results when the same evaluation is performed on its static counterpart. For static fault trees, the combinations of basic events that trigger a top-level event are fixed. Hence, the probability of a top-level event can be determined purely by multiplication/addition (given an AND/OR gate) on the probabilities of the basic events. The probability of the basic events are computed using $1 - e^{\lambda t}$, where λ is the failure rate and t is the duration.

In the dynamic case, where ordering of events is also represented, that same order is then accounted for together the duration for the computation of the top level probability. Consider Figure 9.68 and a duration t. Events E3 and E4 are ordered such that E3 has to occur before E4. Due to this order, event E3 occurs within a time t_1, and then event E4 occurs within a time $t - t_1$. This shows that the time left for event E4 to occur depends on the amount of time spent waiting for event E3 to occur. For computing the probability of E4 occurring, this then more formally expressed as Pr(E4 occurs within $t - t_1$ | E3 occurs within t_1). If the elapsed times are exponentially distributed, then its memory-less property allows to rewrite this to simply Pr(E4 occurs within $t - t_1$).

9.6.7 Criticality Evaluation

Criticality evaluation is performed as an additional step of (dynamic) fault tree evaluation (see Section 9.6.6). The entered Criticality number is used to compute the severity of triggering

the gates and the top-level event. The criticalities are denoted below the probability, namely after the "C = " part of a gate. The criticality is simply the severity times the probability.

Criticality provides an incremental amount of information upon fault tree evaluation. The user has the freedom to choose an appropriate severity classification and the freedom how to interpret the criticalities.

9.6.8 (Dynamic) Fault Tree Verification

(Dynamic) Fault Tree Verification is an advanced and more expressive way to analyse (dynamic) fault trees probabilistically. It uses the property management system to define properties over (dynamic) fault trees and then plot the cumulative distribution function of that property.

Steps

1. Perform (Dynamic) Fault Tree Generation (see Section 9.6.1).
2. Click on the "(Dynamic) Fault Tree Verification" tab (see Figure 9.65).
3. In the properties pane on the left, check a property that has a (dynamic) fault tree associated.
4. Choose a probabilistic pattern. This is the probabilistic property used to analyse the chosen (dynamic) fault tree.
5. Fill in the placeholders. The grammar of a legal placeholder is the nearly the same as for atomic propositions in properties (see Section 7.1). All non-arithmetic operators outlined there are supported. The only difference is the references to ports, data subcomponents, mode variables, mode names, error variables and error state names. They are not allowed. Instead, you can reason over the fault configurations and the gates identifiers. They can be viewed in the fault tree viewer under "View" and "Hide Probabilities" (see Figure 9.70).
6. Click on "Run"

The result is a graph (see Figure 9.69) which has to interpreted similarly as the graphs produced by performability evaluation (see Section 9.5). The main difference is that the graphs are produced with the (dynamic) fault tree as the underlying model. For performability evaluation, the graphs are produced using the SLIM model as the underlying model.

For (dynamic) fault tree verification, the same notes apply as for (dynamic) fault tree evaluation. Thus, we recommend you to ensure that all basic events have a poisson rate associated with it and to check whether the used probabilistic property is underspecified.

9.6.9 Hierarchical Fault Tree Generation

The Hierarchical Fault Tree Generation pane allow safety analysis to be performed on the model structure based on its contracts. This is similar to safety analysis as described in Sections 9.6.1 and 9.6.5, but makes use of the hierarchical composition of the model. For an overview, see Figure 9.71

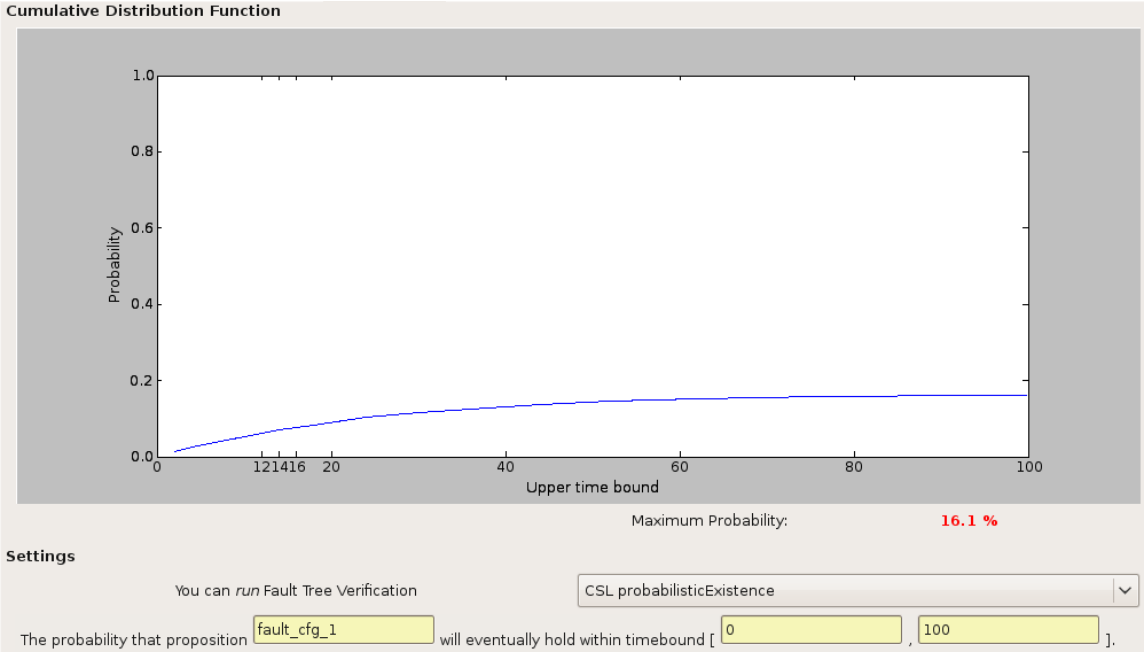


Figure 9.69: Result of (Dynamic) Fault Tree Verification.

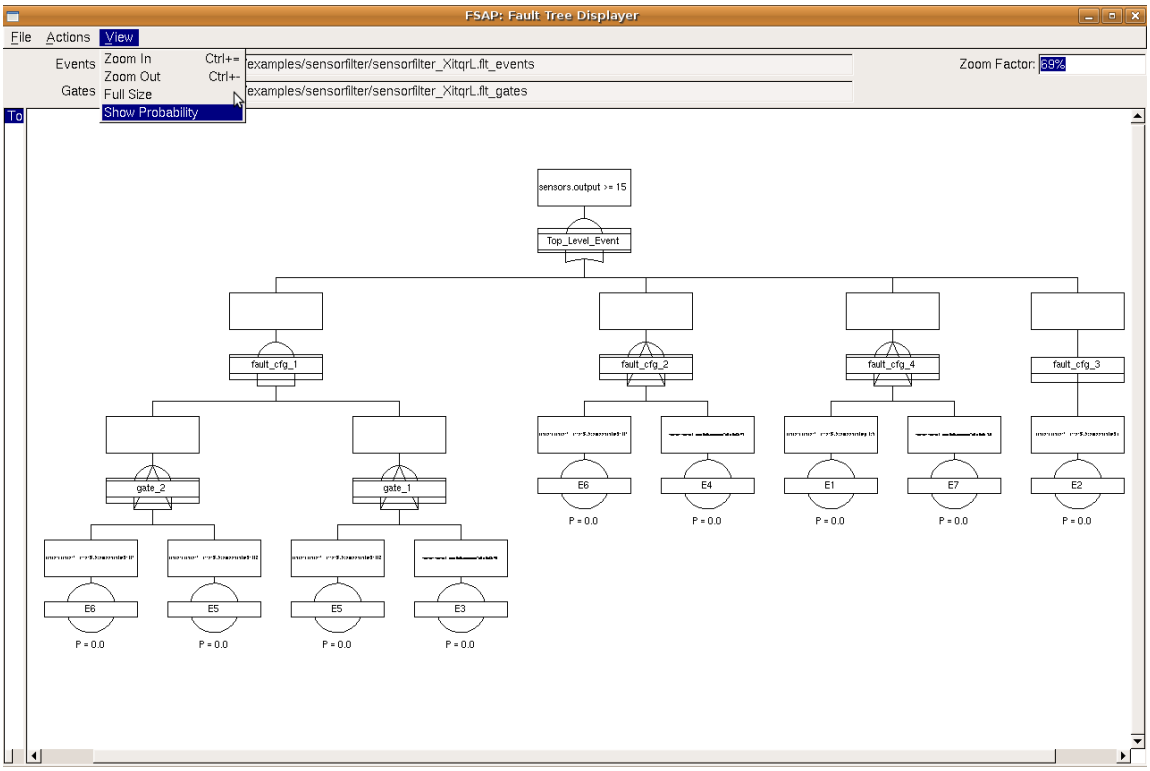


Figure 9.70: Showing the gate and fault configuration labelling in the fault tree viewer.

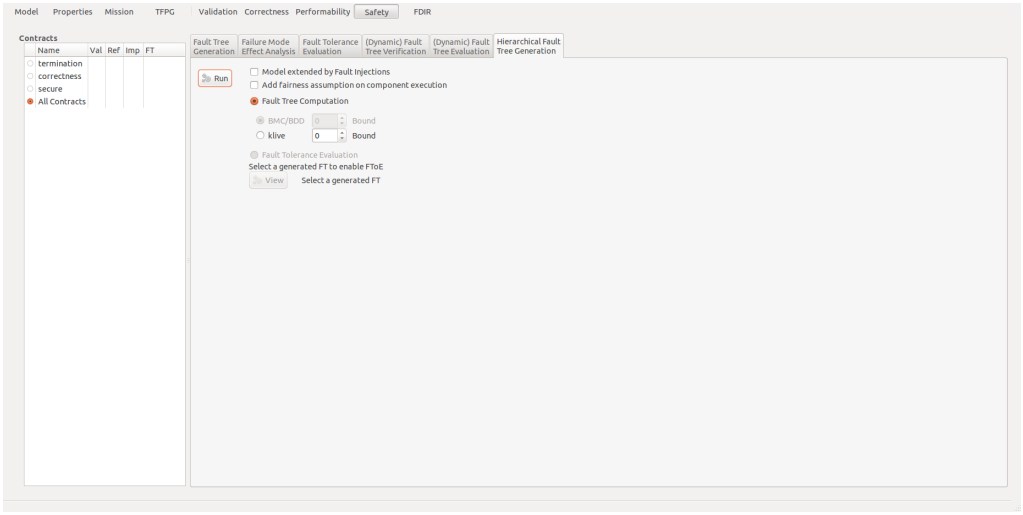


Figure 9.71: Safety

The options *Model extended by Fault Injections* and *Add fairness assumption on component execution* are the same is described in Section 9.2.2. To perform Safety analysis, at the least one contract needs to be selected from the left hand side contract list.

Fault Tree Computation

When selecting *Fault Tree Computation*, a fault tree can be generated from the model. Two possible analysis engines can be selected, *BMC/BDD* and *klive*. See Section 9.4 for their meaning. After running Fault Tree Computation, the column *FT* on the left hand side contract list will show *GEN* to indicate a fault tree has been generated. When selecting a contract with a generated fault tree, the *View* button allows it to be shown.

When selecting *All Contracts* in the contract list, a fault tree will be generated for each contract in the list.

Fault Tolerance Evaluation

The *Fault Tolerance Evaluation* can generate fault tolerance metrics for the selected contract, when this contract has an associated fault tree (the *FT* columns indicates *GEN*). This will calculate the cutsets for the fault tree and their cardinality, as well as indicate the number of faults due to the environment. For more details, see Section 9.6.5.

9.7 FDIR: Fault Detection, Isolation and Recovery

The *FDIR* pane allows the user to analyze the capability of a system to detect, isolate, and recover from faults. It is based on the notion of "observable/alarm" signals in the model. This analysis groups four types of checks:

1. *Fault Detection Analysis* that allows the user to check whether a fault can be detected;
2. *Fault Isolation Analysis* that allows the user to check whether a fault can be isolated;

3. *Fault Recovery Analysis* that allows the user to check whether the system can keep working, even if a fault has occurred; and

4. *Diagnosability Analysis* that allows the user to obtain information on the diagnosability of the system.

Figure 9.72 shows the main view of the *Fault Detection, Isolation and Recovery* pane.

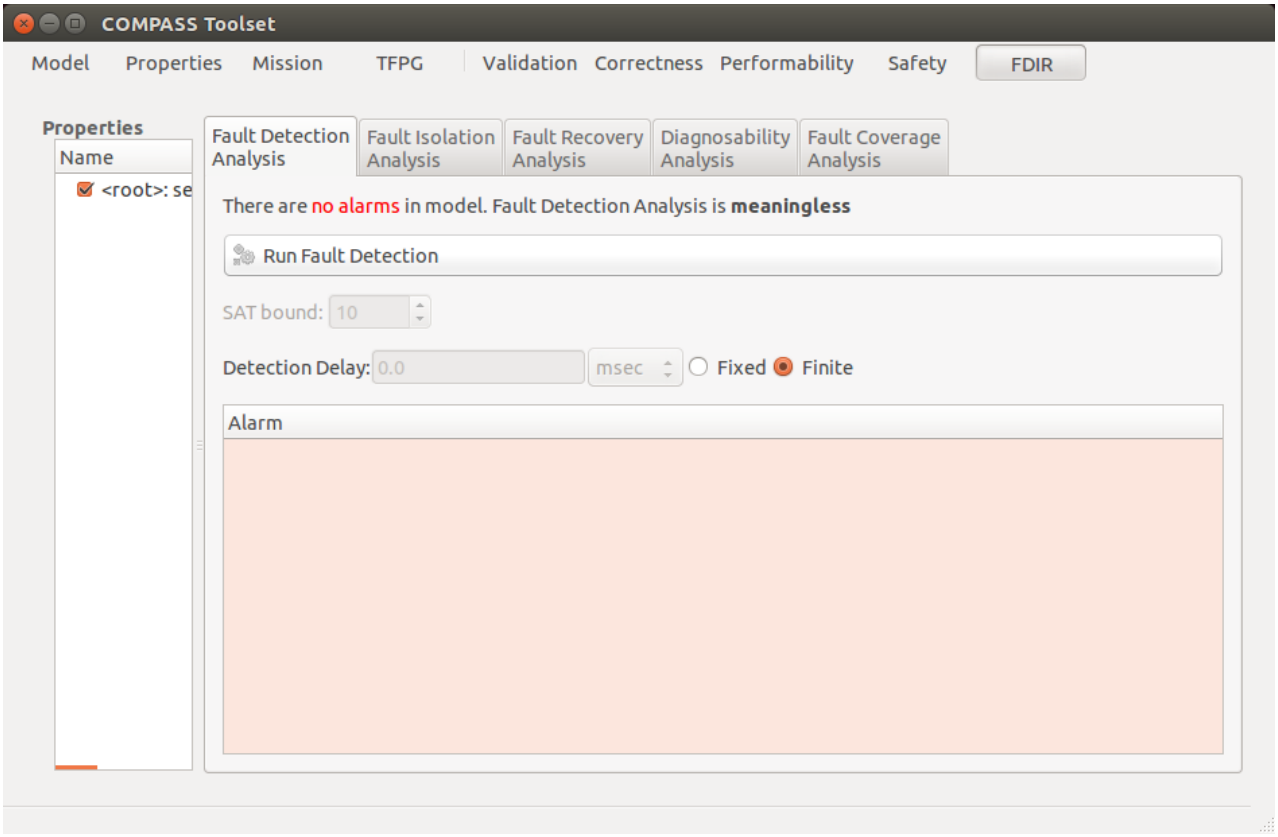


Figure 9.72: The FDIR main pane

By default, the *Fault Detection Analysis* view is displayed. Below we describe each of the panes in more detail.

9.7.1 Fault Detection Analysis

This pane allows the user to find out which alarms (if any) will be eventually raised whenever the selected property becomes true. The alarms correspond to the possible detection means for the property. The property must be purely propositional. The corresponding pane is the one shown in Figure 9.72.

Steps

1. Select the SLIM model from the *Model* pane.

2. Specify one or more fault injections.

3. Specify a property to be checked in the *Properties* pane.
4. Select the property to check in the *Properties* section of the *Fault Detection Analysis* pane (compare Figure 9.73).
5. Then, clicking on the button *Run Fault Detection* fills the list of alarms, as shown in Figure 9.74. Each alarm is one possible *detection means* for the fault expressed by the chosen property.

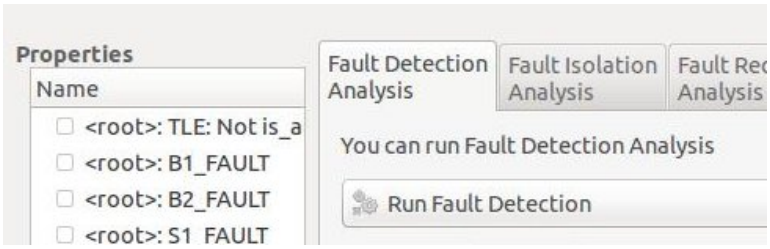


Figure 9.73: The fault detection analysis main pane enabled

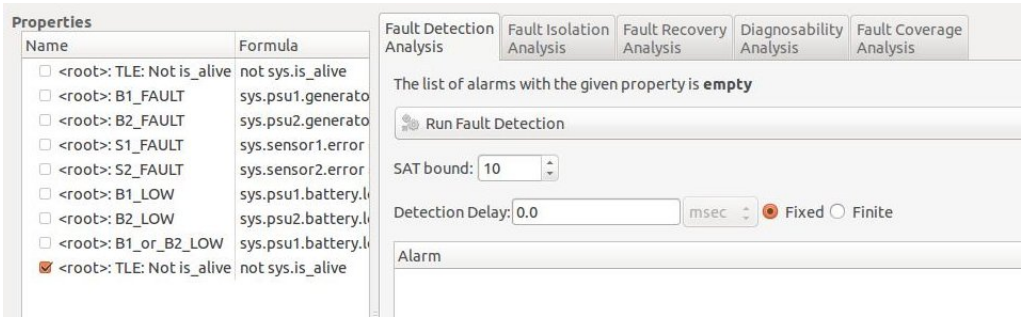


Figure 9.74: The fault detection analysis pane when an empty set of sensor is computed

In the example in Figure 9.74, no alarm is computed (that is, the fault is not detectable); if we try to change fault, then the system will respond that the `mon.alarm_battery`, `mon.alarm_battery1` and `mon.alarm_battery2` sensors will be raised (that is, e.g. `mon.alarm_battery` is a possible detection means) if the property becomes true, as shown in Figure 9.75.

9.7.2 Fault Isolation Analysis

Fault isolation analysis creates one or more fault tree(s), one for each alarm signal in the model. For each alarm, a fault tree will be generated in order to show the combinations of events that may cause the fault. In case of perfect isolation, only one combination (typically with cardinality one) will be generated. The analysis works without selecting any property (note, in fact, that the *Properties* pane is disabled), as shown in Figure 9.76.

Steps

1. Select the SLIM model from the *Model* pane.
2. Specify one or more fault injections.

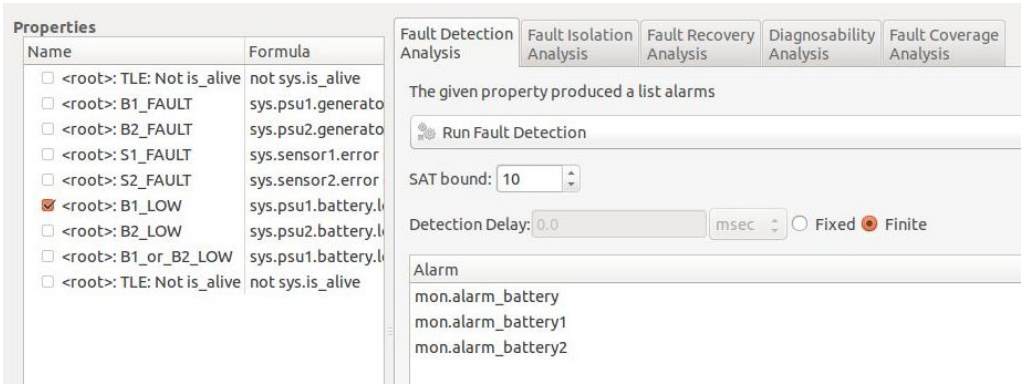


Figure 9.75: The fault detection analysis pane when some sensor will be raised

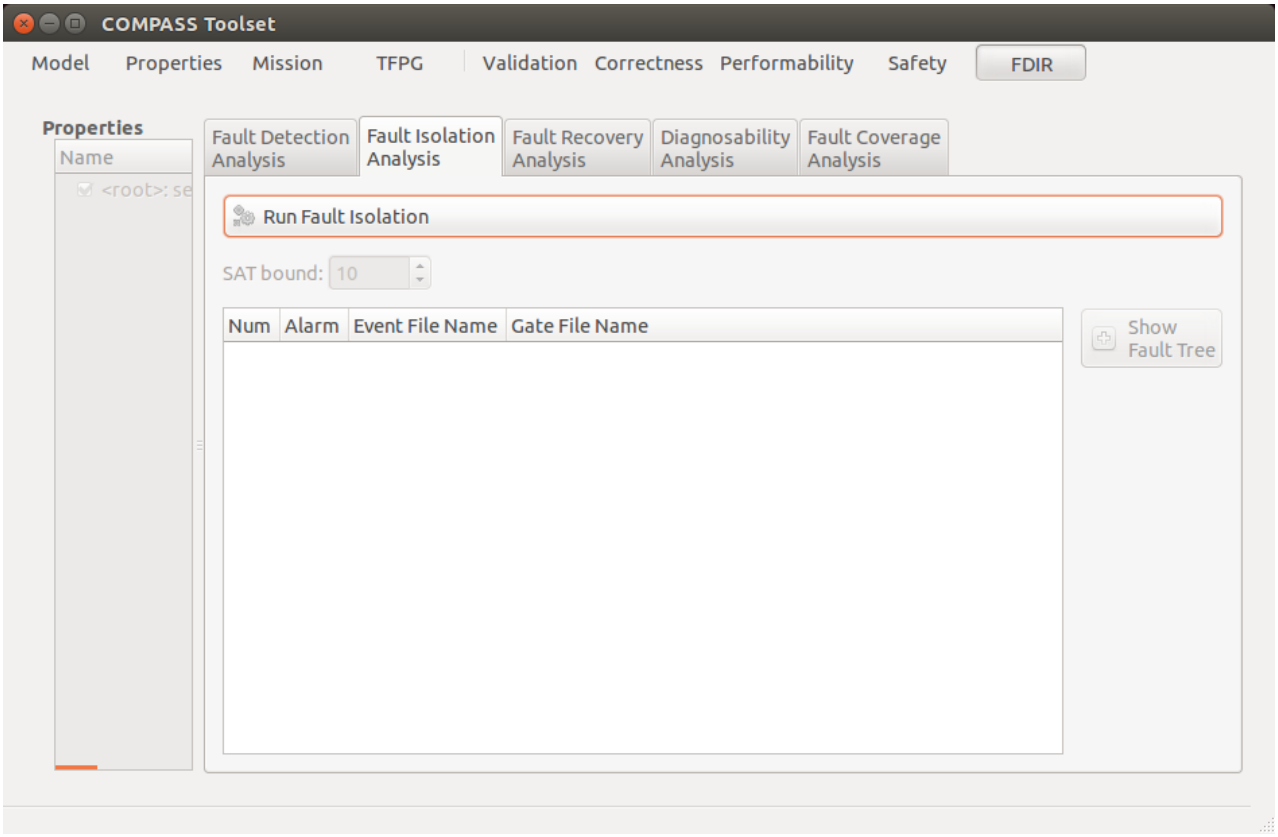


Figure 9.76: The *Fault Isolation Analysis* main pane

3. After clicking on the *Run Fault Isolation* button, the table shown in Figure 9.77 is filled with the names of the events and gates files (one for each fault tree).

4. By double clicking on a row, or with a single click on a row and a single click on the button *Show Fault Tree*, the fault tree displayer will be automatically invoked with the event and gate files in the selected row, and will show the selected failure as in Figure 9.78.

In this example, for the `monitor.detected` signal there are several ways in which it can be raised, as shown in the tree. Hence, fault isolation is not perfect.

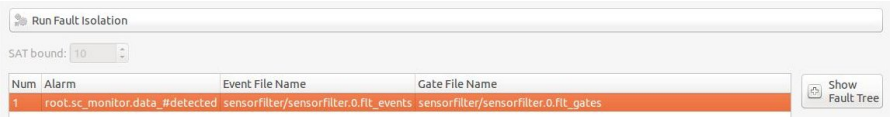


Figure 9.77: The fault isolation analysis pane results

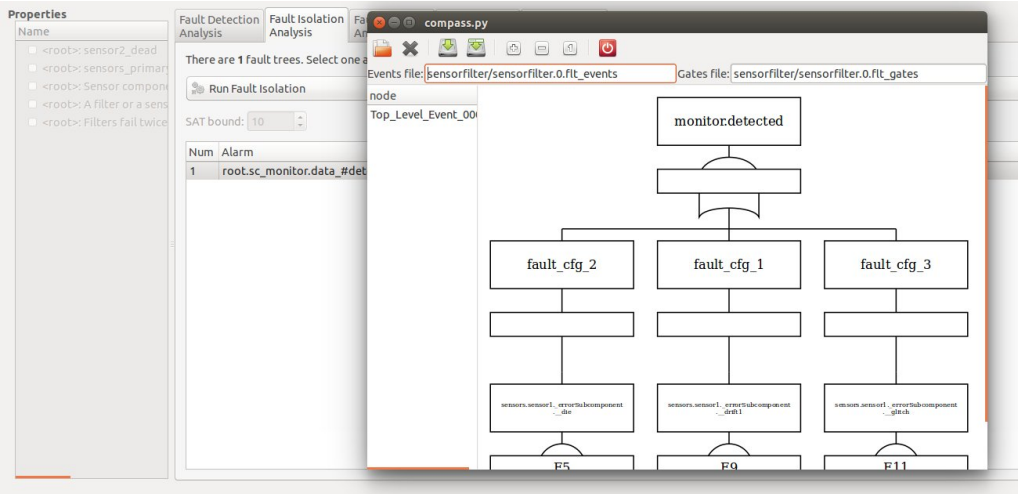


Figure 9.78: The fault tree displayer invoked in the fault isolation analysis

9.7.3 Fault Recovery Analysis

This pane provides the user information about the recoverability from possible faults. It uses a model checker to prove if the selected recoverability property holds or not (the options are the same as for model checking, see Section 9.4.4 for further information). The main view looks like in Figure 9.79.

Steps

1. Select the SLIM model from the *Model* pane.
2. Specify one or more fault injections.
3. Specify a property to check in the *Properties* pane. After selecting one or more properties, the button *Run Fault Recovery* will be activated, as shown in Figure 9.80.
4. After clicking on the button *Run Fault Recovery*, the model checking view will be shown. (See Section 9.4.4 for more information about the filters and how to read the trace).
   (a) If the fault is not recoverable the view looks like the one in Figure 9.81. In this case the trace shows an execution of the model in which the property is not satisfied
   (b) If the property is satisfied instead, the view looks like the one in Figure 9.82.

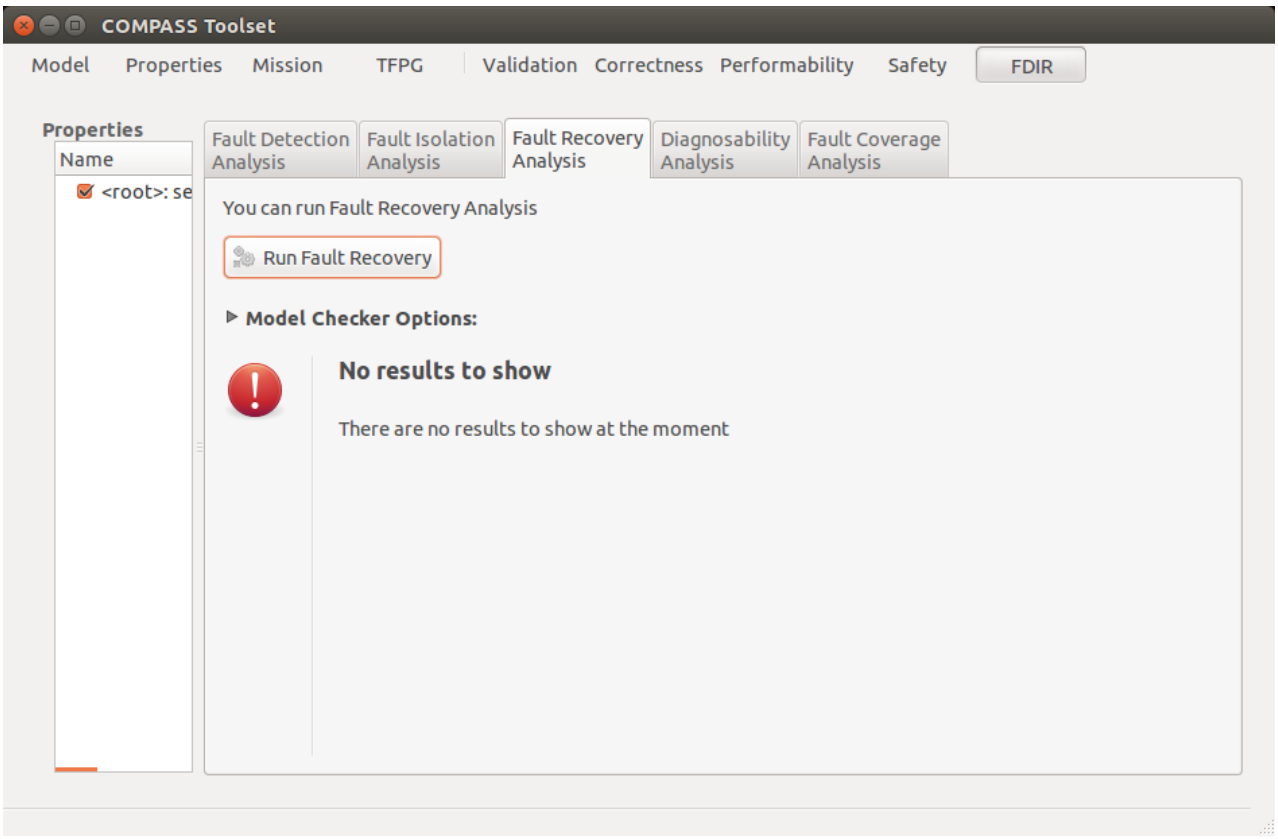


Figure 9.79: The fault recovery analysis main pane

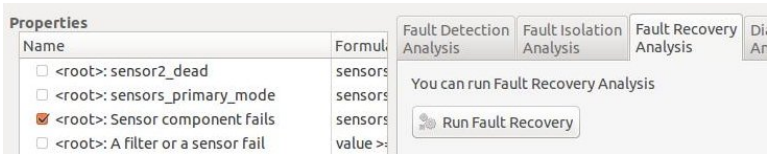


Figure 9.80: The fault recovery analysis enabled

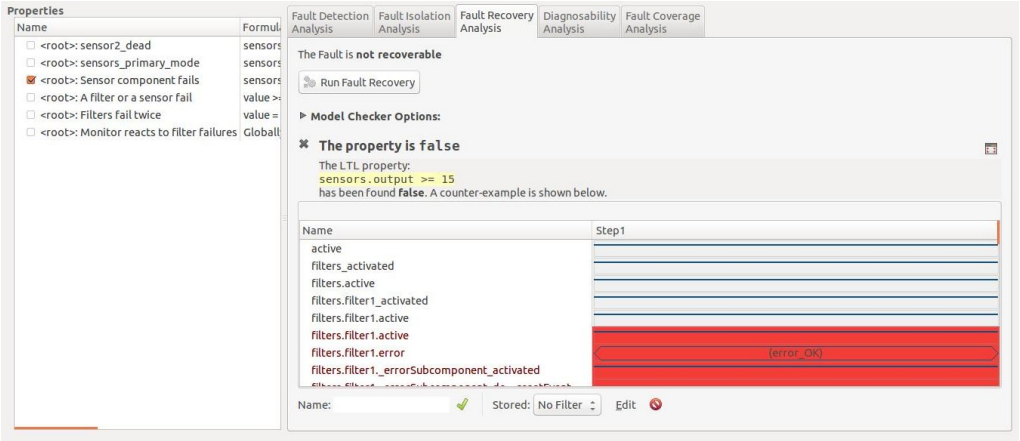


Figure 9.81: The fault recovery analysis pane when the fault is not recoverable

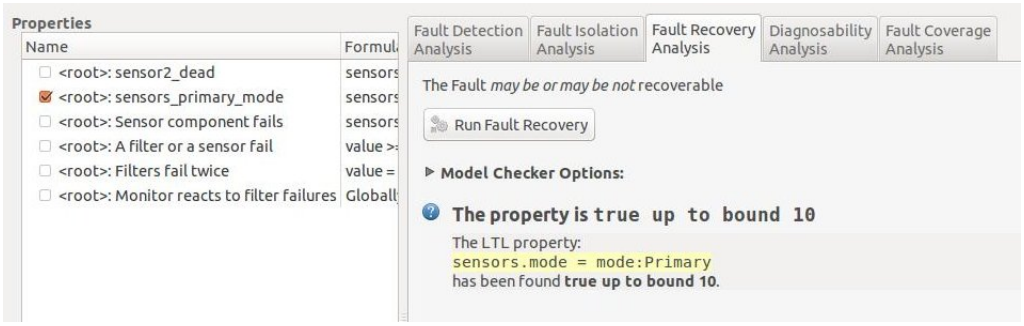


Figure 9.82: The fault recovery analysis pane when the fault is recoverable

9.7.4 Diagnosability Analysis

The safety engineer can use the diagnosability analysis functionality in COMPASS to obtain information on the expressiveness of sensor data. Simply put, one is interested to know whether the observables specified in the SLIM model files are enough to infer certain properties, specifically if they are adequate to diagnose the presence of faults.

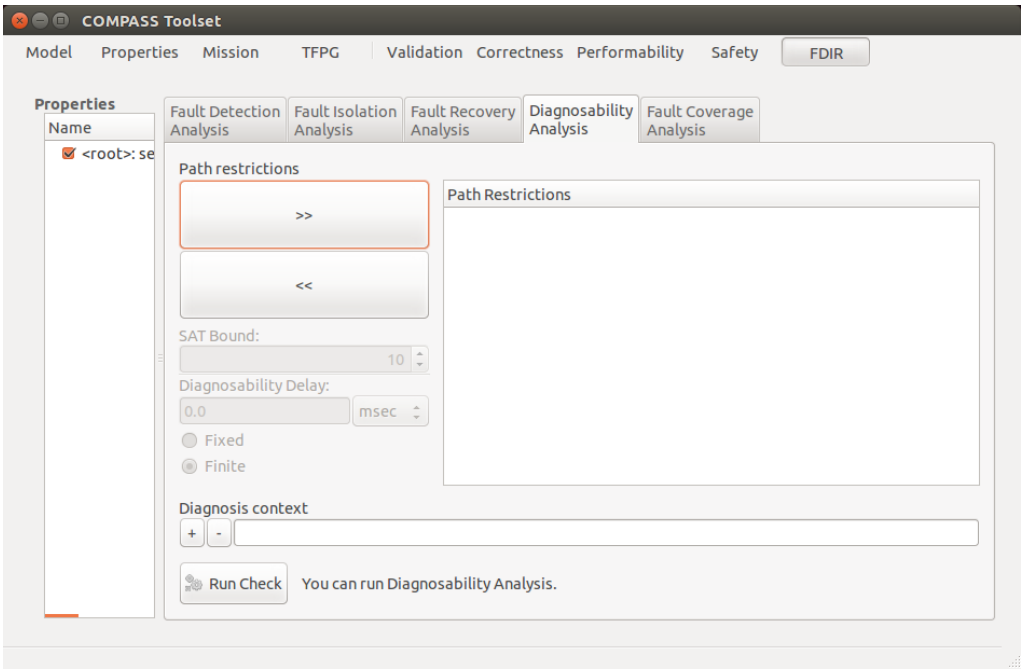


Figure 9.83: Diagnosability analysis view

Preconditions These are the preconditions for running diagnosability analysis.

1. Load the nominal and error SLIM models.
2. Perform the desired fault injections.
3. In the *Properties* pane, specify one or more properties to check in the following steps.

Note all properties to be used in diagnosability analysis have to be propositional.

Steps

1. Click on *FDIR*, then on *Diagnosability Analysis*.
2. (optional) Choose one or more path restrictions by clicking the corresponding property in the left pane, and adding it to the list in the center by using the right-arrow button; the left-arrow button can be used to remove path restrictions; furthermore, they can be rearranged by drag-and-drop.
3. (optional) Choose one diagnosability property context by clicking one in the list of properties, and clicking the "+" button. Use the "–" button to clear the context.
4. (optional) In the case of a hybrid model, bounded model checking will be used. The default value for the bound is 10, but this value can be changed by the corresponding setting box in the *Diagnosability Analysis* pane.
5. Choose the diagnosability property by clicking the desired entry in the *Properties* list.
6. Click *Run Check*. An activity bar at the bottom of the screen will show the ongoing activity, and the appearing *Stop* button can be used to interrupt the process.

Result The result will be announced in a text message right next to the *Run Check* button. This can be either *diagnosable*, *not diagnosable* (plus counterexample), or *unknown* (in case of hybrid models where no counterexample has been found within the given bound).

In case a property was found to be non-diagnosable, a trace window will be opened containing two trace views, which constitute the counterexample for diagnosability. Both traces exhibit equal observational data in synchronous states. There will also be two synchronous states which both satisfy the context property, but one of them satisfies the given diagnosability property, and the other one does not (but satisfies its negation). Furthermore, if any path restrictions were specified, each of them appears in the given total order in both traces preceding the aforementioned two synchronous states; notice that these path restrictions do not need to be synchronous.

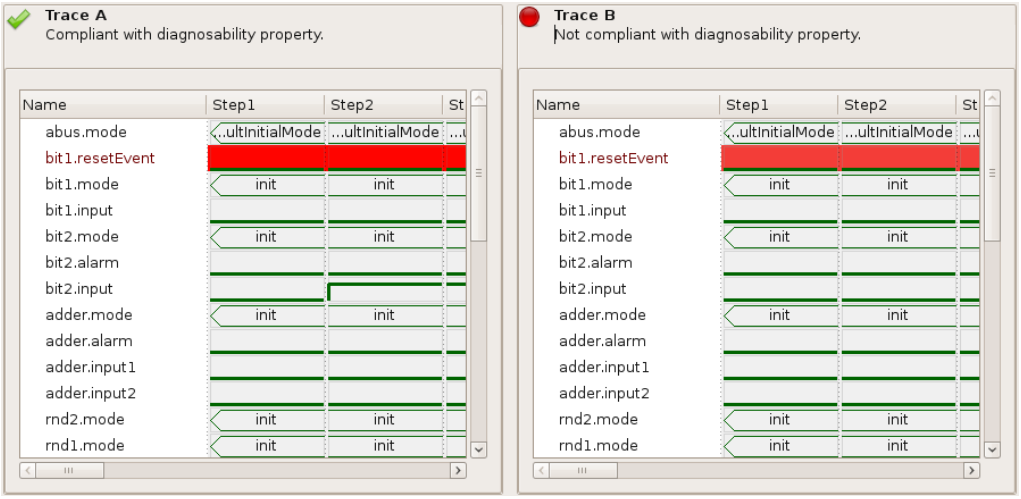


Figure 9.84: Counterexample for diagnosability

Example For demonstration purposes, let us assume we are working on the Adder model that can be found in the example files. Note that only `bit1.alarm1`, `bit2.alarm1`, and `adder.alarm1` are observable.

1. We load the nominal model `Adder.slim` and the error model `Adder_err.slim`.
2. We assume that component `Bit1` can break and perform a fault injection such that on error state `inverted`, `bit1.output` becomes "`not input`", and `bit1.alarm1` becomes `true`.
3. With this background, we are interested to see if we can diagnose a corrupt `adder.output` value. First, we define a context property in the property manager to specify what case we are interested in. Let us require that both `rnd1` and `rnd2` deliver a value of 1; also, our analysis is limited to states in which a computation is performed. We therefore define the property "`run and rnd1.output and rnd2.output`".
4. The context implies that if `bit1` is not broken, the adder output should be 0. Now the question is whether we are able to diagnose that the value of `adder.output` is 1, when it should be 0. We thus define the property `adder.output`.
5. At this point, we go to the *diagnosability analysis* window, select the context property, click the plus sign for specifying the property context, select the diagnosability property, and click the *Run Check* button. The result will be displayed in the middle of the screen.
6. If we change the context to "`rnd1.output and rnd2.output`", we will see that the property is not diagnosable anymore. Two synchronous traces will be shown which exhibit the same observations in each state; at one synchronous state, both satisfy the property context, but one satisfies the diagnosability property while the second one does not – it satisfies exactly the negation of the diagnosability property. This is a counterproof for diagnosability.
7. Path restrictions can be specified by selecting properties and then adding them to the list in the middle of the window by clicking the right-arrow button. The order of path restriction properties can be modified by drag-and-drop. Deletion works by selecting a property and clicking the left-arrow button. In traces returned by COMPASS you will notice that the states containing the conflict are always preceded in the specified total order by those path restrictions.

Hybrid models One must be careful when working with assignments to observable variables (Boolean data ports or data subcomponents) that refer to variables with a continuous domain. For instance, if a certain continuous value x goes from -1 to 1, it crosses the value 0. If one of our formulas depends on the fact that $x = 0$, and this x is tested in an assignment to an observable data port p, the observation that $x = 0$ might go by unnoticed.

The reason for this is that the discretization of the continuous variable for delivery to the data port happens non-deterministically at random intervals, which means that it is possible but not guaranteed that p detects the value $x = 0$.

If this however is explicitly desired, discretization at $x = 0$ must be manually specified in the SLIM model. This can be accomplished, for example, by splitting the mode that defines the dynamics of x into two sub-modes where the first is taken for non-positive values of x (by

requiring the invariant $x <= 0$), and is left when the value $x = 0$ is reached. To this aim, the corresponding transition to the second sub-mode is guarded by the condition $x = 0$, and sets p to the value true.

9.7.5 Fault Coverage Analysis

This pane provides the user information about the level of coverage of possible faults, depending on whether they can be detected/recovered from. It uses a model checker to prove if the selected detection/recovery properties hold or not. The main view is shown in Figure 9.85.

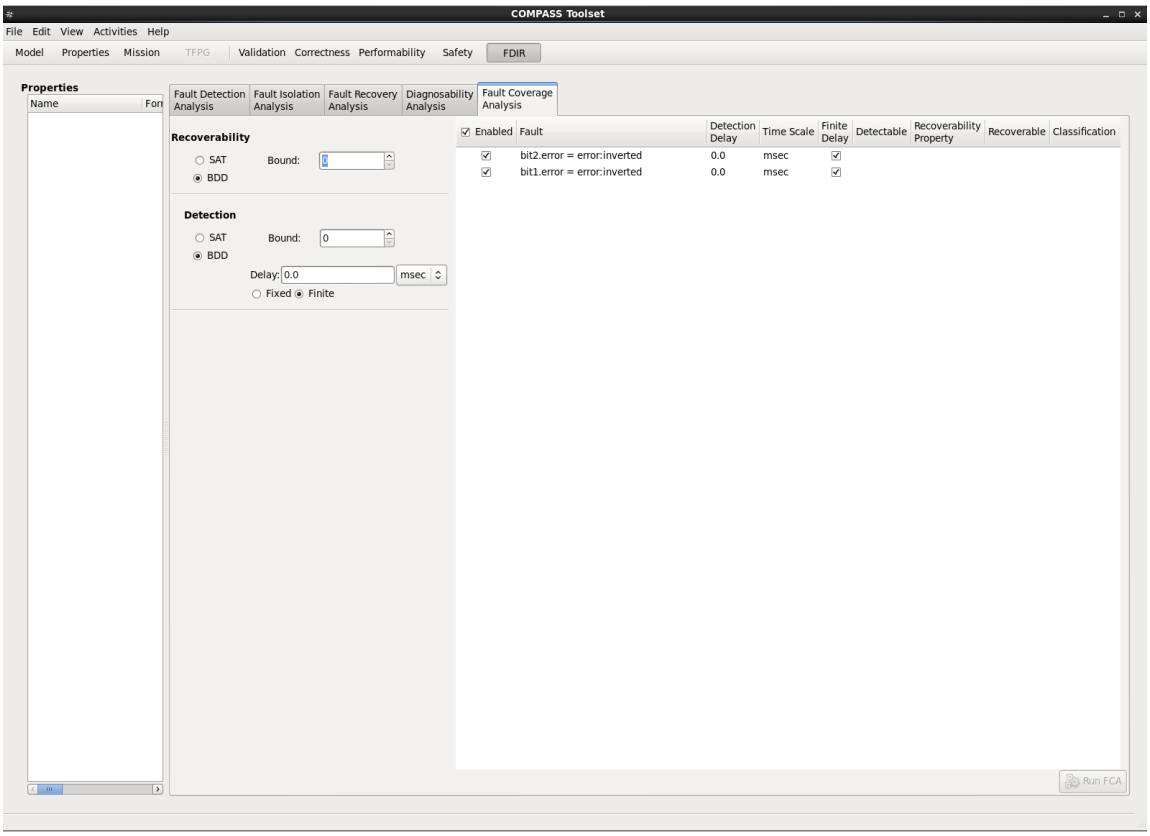


Figure 9.85: The fault coverage analysis main pane

Steps

1. Select the SLIM model from the *Model* pane (eg. Adder_recov.slim and Added_err.slim).
2. Specify one or more fault injections (e.g. only the fault bit1.output := input = false).
3. Select the *Fault Coverage Analysis* Tab
4. Enable one or more *faults* (e.g., only the fault bit1.error = error:inverted)
5. Select a *recoverability property* for each fault. After selecting one or more properties for fault (e.g., bit1_inverted), the button *Run FCA* will be activated.

6. Set the BDD or the SAT Bound options for recoverability and detection (e.g. SAT Bound=50)

7. Set the Detection Delay as:

   (a) **`Finite`**: specifies that the fault must be detected within a finite bound

   (b) **`Fixed`**: it is a real value the maximum detection bound for the given fault. It is also possible to specify the time scale unit:

      i. msec
      ii. sec
      iii. min
      iv. hour
      v. day

8. After clicking on the button *Run FCA*, when the analysis is completed, the results of Fault Coverage Analysis will be shown (as in Figure 9.86)

   (a) Fault detection analysis is run to determine if the fault can be detected by any existing alarm. If the fault is not always detected, then it is classified as *Uncovered*.

   (b) Fault recovery analysis is run (using the provided recoverability property), to determine if the fault can be recovered from.

      i. If it can always be recovered from, it is classified as *Covered*.
      ii. Otherwise, it is classified as *Uncovered*.

Remark: Classification as 'partially covered' is not supported – it would amount to deciding that a given fault is recoverable only in specific configurations. Implementation of this feature is left for future work.

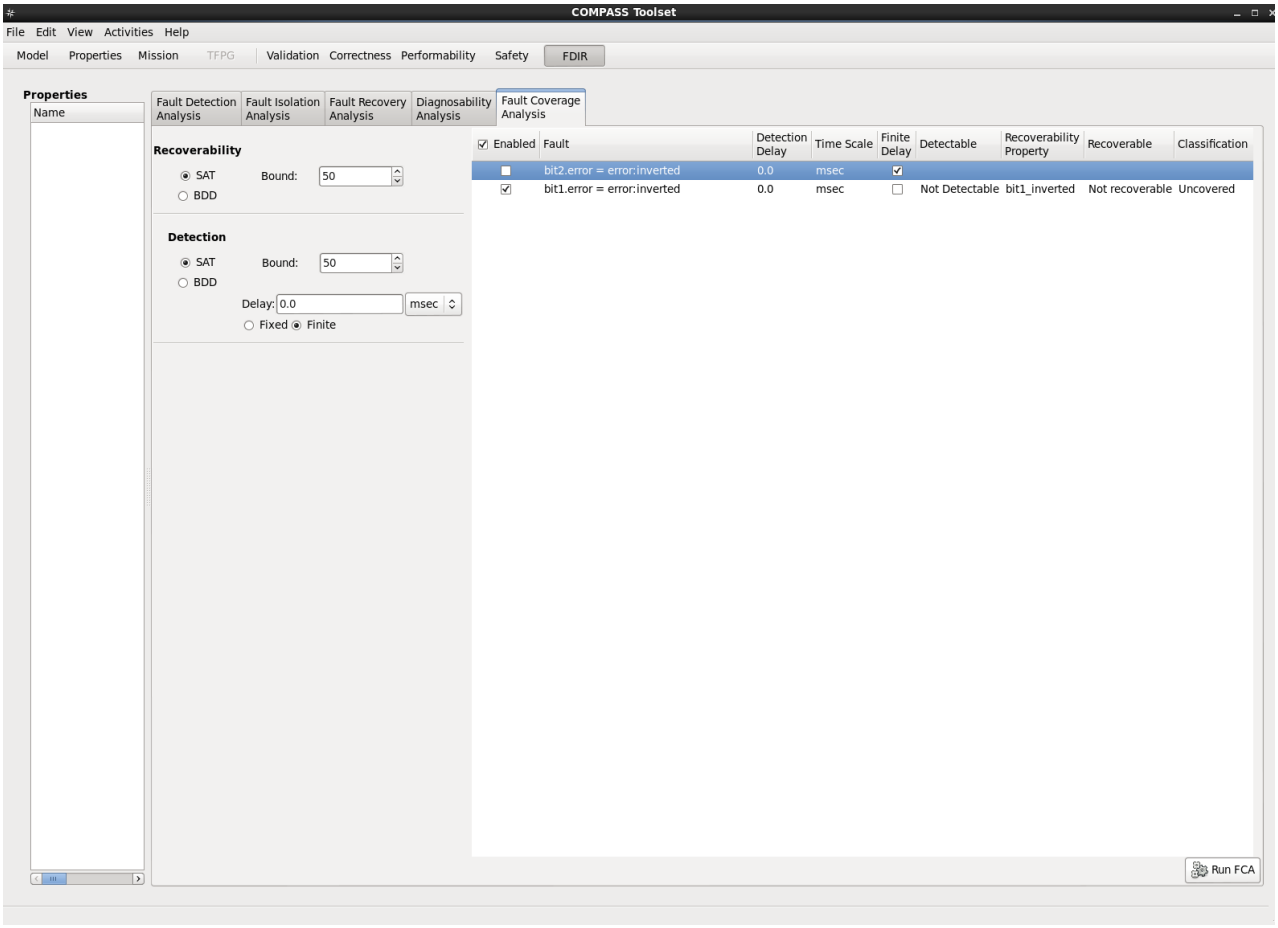


Figure 9.86: The fault coverage analysis results

Chapter 10

Support

Official support is provided within timeframe and clauses set by and under Contract No. 4000115870/15/NL/FE/as. In all other cases there is not warranty for this toolset and all of its documentation.

The primary channel for support is email: support requests can be send to compass-support@lists.rwth-aachen.de.

Bibliography

[1] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. The COMPASS approach: Correctness, modelling and performability of aerospace systems. In *Proc. 28th Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP 2009)*, volume 5775 of *LNCS*, pages 173–186. Springer, 2009.

[2] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pages 70–87, 2011.

[3] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. IC3 Modulo Theories via Implicit Predicate Abstraction. In *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*, pages 46–61, 2014.

[4] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Verifying LTL Properties of Hybrid Systems with K-Liveness. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 424–440, 2014.

[5] Koen Claessen and Niklas Sörensson. A liveness checking algorithm that counts. In *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, pages 52–59, 2012.

[6] Edmund Clarke, Daniel Kroening, Ofer Strichman, and Joel Ouaknine. Completeness and complexity of bounded model checking. In *5th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 85–96, 2004.

[7] COMPASS Tutorial. Technical report, COMPASS Consortium, 2016.

[8] Catalogue of system and software properties. Technical Note D1-2, Issue 4.7, COMPASS Project, June 2016.

[9] T.A. Henzinger. The theory of hybrid automata. In *IEEE Symp. on Logic in Computer Science (LICS)*, pages 278–292. IEEE CS Press, 1996.

[10] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In L. Zuck, P. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *4th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309. Springer Verlag, January 2003.

[11] Timo Latvala, Armin Biere, Keijo Heljanko, and Tommi A. Junttila. Simple bounded ltl model checking. In Alan J. Hu and Andrew K. Martin, editors, *FMCAD*, volume 3312 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 2004.

[12] Slim 3.0 - syntax and semantics. Technical Note D1-2, Issue 4.7, COMPASS Project, June 2016.

[13] Stefano Tonetta. Abstract Model Checking without Computing the Abstraction. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, pages 89–105, 2009.

Appendix A

CLI scripts

The COMPASS toolset also comes as a set of scripts which can be run from the command line. These CLI (Command Line Interface) scripts provide a means to interact with the toolset using the console. It has the added advantage of automating analysis using shell scripts.

A.1 Scripts

In the following explanations the scripts are run from the top-level directory. In the examples all the paths to files will be given with respect to that top-level directory. For example, all the scripts themselves are placed in directory **scripts**. A user is assumed to change the working directory to the top-level directory (using e.g. `cd`) before running the scripts.

In general the scripts take one or more SLIM files with nominal models, error models and FDIR models as its arguments. For more details on allowed command line options see Section A.1.29 which describes advanced options that are common to all the scripts. For individual scripts the descriptions can be obtained by running the script with option **`--help`**, e.g.

```
$> scripts/check_syntax.py --help
```

The scripts are Python files and require a `python` interpreter to be run. The following subsections describe the available scripts in more detail.

A.1.1 Syntax Check

The simplest example of running a (test) script is

```
$> scripts/check_syntax.py \
    documentation/examples/adder/adder_discrete/adder.slim \
    documentation/examples/adder/adder_discrete/adder_err.slim
```

which just reads the SLIM files for the Adder example and checks it for syntactic correctness. Since the file is syntactically correct the script outputs

```
File 'documentation/examples/adder/adder_discrete/adder.slim' parsing: OK
```

A.1.2 Model Checking

Verify whether a model (optionally extended with a fault and/or FDIR model) satisfies a (qualitative) property		
Inputs	M	an operational model
	FM	a fault model (optional)
	FDM	an FDIR model (optional)
	P	a set of properties
Outputs	$true\|false\|unknown$	truth value for each property $\phi \in P$
	Tr	A counter-example for false properties
For each property in P, *true* shall be returned if the property holds, or else a counter-example trace Tr in XML should be produced		

```
$> scripts/check_properties.py \
    [--property-file=<properties_file_name>] \
    [--bmc [--bmc-length=<length>]] \
    [--ocra] [--klive-bound] \
    [-i] file1.slim ... fileN.slim
```

Option –**bmc** forces to use Bounded Model Checking when performing LTL model checking, while option –**bmc-length** specifies the max length to be used when performing BMC

Results are printed out to stdout, and possibly generated traces are dumped into XML files (one file each trace).

For example:

```
$> scripts/check_properties.py -i \
    -p documentation/examples/adder/adder_discrete/adder.propxml
    documentation/examples/adder/adder_discrete/adder.slim \
    documentation/examples/adder/adder_discrete/adder_err.slim

Model Checking: Formula 'never adder error: Globally, it is never (...)' is false
Counterexample in file 'adder/adder_trace1.xml'
```

In the example, the first CTL formula has been found to be false, and a corresponding counter-example has been generated in `adder/adder_trace1.xml`.

To see the content of a trace, the **trace viewer** can be used:

```
$> scripts/view_trace.py <trace-file.xml>
```

By default, model checking uses a technology based on Binary Decision Diagrams (BDDs). BDDs can be very effective and can be used for both CTL and LTL formulae. However, under some circumstances the size of the model that is being checked can cause a blow up in memory or in time.

In this cases, when dealing with LTL formulae only, it is possible to exploit a different technology called Bounded Model Checking (BMC). BMC can be used to prove that a formula is false, and in some condition it can also prove that a formula is true. This makes BMC ideal when searching for bugs, to show that assumptions about the model (expressed as LTL formulae) are actually false.

BMC is based on a bound that is incremented up to a maximum number that is called *problem length*, and that corresponds to the maximal number of execution steps of the system under consideration. Given a problem length K, if a LTL formulae is found **false** at bound j, $0 <= j < K$, a counter-example with length j is returned. The formula might also be found **true**, and in this case there will be no trace available. If the problem length K is reached with no result found, the result is stated to be **unknown**. This means that either the property is true, or it may be found false at a higher bound.

To use BMC when checking LTL formulae the option **`--bmc`** should be given. To optionally set a different problem length use option **`--bmc-length`** (default is 10).

Notice that when option **`--bmc`** is specified, BMC will be used only for LTL formulae, whereas for CTL formulae the BDD technology will be used.

```
$> scripts/check_properties.py \
    --bmc --bmc-length=20 -i
    documentation/examples/adder/adder_discrete/adder.slim \
    documentation/examples/adder/adder_discrete/adder_err.slim

Model Checking: Formula 'G !run and rnd1.output = false and (...)'  is false
Counterexample in file 'Adder_trace2.xml'
Model Checking: Formula 'G !run and rnd1.output = false and (...)'  is unknown
```

In case we want to run MITL (Metric Interval Temporal Checking) checking, we can specify the max bound for kliveness using the option **`--klive-bound`**.

This script can also be used to check the OCRA monolithic implementation; in this case option **`--ocra`** must be used.

A.1.3 Model Simulation

Produce a simulation trace of an operational model (optionally extended with a fault model and/or an FDIR model)		
Inputs	M FM FDM C	an operational model to be simulated a fault model (optional) an FDIR model (optional) a set of user constraints on the simulation
Outputs	Tr	a simulation trace
Using model checking techniques, the COMPASS toolset shall produce a simulation trace of the given operational model (optionally extended with a fault model and an FDIR model). The trace shall conform to a set of simulation constraints. The output trace shall be generated in XML format.		

```
$> scripts/simulate_with_bmc.py \
    [--sim-length=<length>] [-i] \
    -p <properties_file.xml> \
    file1.slim ... fileN.slim
```

One simulation trace XML file will be created.

```
$> scripts/simulate_with_bmc.py \
    documentation/examples/adder/adder_discrete/adder.slim \
    documentation/examples/adder/adder_discrete/adder_err.slim

Trace of simulation can be found in 'adder/adder_trace3.xml'
```

The resulting trace will have a length l, which is equal to the value specified with option **`--sim-length`** (default is 10).

To see the content of a trace, the **trace viewer** can be used, .e.g.:

```
$> scripts/view_trace.py adder/adder_trace3.xml
```

The simulation is obtained by randomly taking l transitions. It is possible to constrain the simulation by specifying one or more propositional formulae as constraints. The given formulae are conjuncted and considered as an invariant. For example if the formulae are `x > 2` and `sc.y = 5`, the invariant formula `x > 2 and sc.y = 5` is built, and the simulation will select only those paths that do not contradict it.

If there exists no valid trace with the requested length satisfying the given invariant formulae, then no trace is produced. This can happen du to inconsistent invariants, or because the given constraints are too strong with respect to the model.

Propositional formulae for constraints can be specified through a properties XML file.

In this example Adder with error injection is simulated with all constraints found in given properties file, and given simulation length:

```
$> scripts/simulate_with_bmc.py \
     --sim-length=5 -i \
    documentation/examples/adder/adder_discrete/adder.slim \
    documentation/examples/adder/adder_discrete/adder_err.slim

Trace of simulation can be found in 'adder/adder_trace4.xml'
```

A.1.4 Deadlock Checking

Check whether there are deadlocks in the model		
Inputs	M	SLIM model
Outputs	m	A message saying whether the model contains deadlocks or not
Using model checking techniques, the COMPASS toolset looks for deadlocks in the model and warns the user about it.		

The following is an example of running the cli-script:

```
$> scripts/check_deadlocks.py \
    documentation/examples/adder/adder_discrete/adder.slim \
    documentation/examples/adder/adder_discrete/adder_err.slim

FSM is deadlock free
```

A.1.5 Fault Tree Generation

Generates a fault tree given an operational model extended with a fault model, optionally extended with an FDIR model and a property representing the top level event.		
Inputs	M FM FDM P	an operational model a fault model an FDIR model (optional) a property (top-level event)
Outputs	Fault Tree	two files, one with events and one with gates
Using model checking techniques, the COMPASS toolset generates a fault tree starting from an operational model extended with a fault model, optionally extended with an FDIR model, and a property representing the top level event. The fault tree is returned as two files, one with events and another with gates, in the format suitable for the fault tree viewer.		

```
$> scripts/compute_fault_tree.py -i \
    -p <properties_file.xml> \
    [--property-num=N] \
    [--dynamic] \
    [--output-xml-and-rates] \
    file1.slim ... fileN.slim
```

The properties in the properties file have to be purely propositional. By default the fault tree is generated for the first property, but with option **`--property-num`** it is possible to specify which property exactly to deal with. Option **`--dynamic`** allows to generate a dynamic fault tree, and option **`--output-xml-and-rates`** is used to tell the script to output the fault tree in XML format and to output the rates of the basic events, which can be used for fault tree verification and fault tree evaluation (see next sections). Running the above script generates two files with extensions `.flt_events` and `.flt_gates`, respectively. For example:

```
$> scripts/compute_fault_tree.py \
    -i --property-num 2 \
    -p documentation/examples/adder/adder_discrete/adder.propxml \
    documentation/examples/adder/adder_discrete/adder.slim \
    documentation/examples/adder/adder_discrete/adder_err.slim
```

generates files `adder/adder.flt_events` and `adder/adder.flt_gates`. To see the content of a fault tree, the **fault tree viewer** can be used:

```
$> scripts/view_ft.py \
    --events-file <events-file.flt_events>
    --gates-file <gates-file.flt_gates>
```

For the above example the fault tree can be viewed using the corresponding cli-script:

```
$> scripts/view_ft.py \
    --events-file adder/adder.flt_events \
    --gates-file adder/adder.flt_gates
```

A.1.6 Failure Modes and Effects Analysis

Generate an FMEA table given an operational model, extended with a fault model, optionally extended with an FDIR model, a set of fault configurations and a set of properties.		
Inputs	M FM FDM FC P	an operational model a fault model an FDIR model (optional) cardinality of a set of fault configurations a set of properties
Outputs	FMEA Table	Failure Modes and Effects Analysis Table
Using model checking techniques, the COMPASS toolset generates an FMEA table starting from an operational model extended with a fault model, optionally extended with an FDIR model, cardinality of a set of fault configurations, and a set of properties.		

```
$> scripts/compute_fmea_table.py \
    -i [--faults-cardinality=N] \
    file1.slim ... fileN.slim
```

The properties in the properties file have to be purely propositional. Option –**faults-cardinality** allows to specify the cardinality of fault configurations, which is by default 1. Running the above script generates a file with extension `.fmea`. For example:

```
$> scripts/compute_fmea_table.py \
     -i --property 'bit1_inverted' \
     -p documentation/examples/adder/adder_discrete/adder.propxml \
    documentation/examples/adder/adder_discrete/adder.slim \
    documentation/examples/adder/adder_discrete/adder_err.slim
```

generates FMEA Table file `adder/adder.fmea`.

A.1.7 Diagnosability Check

Check whether a model is diagnosable with respect to a diagnosability property.		
Inputs	IM	an integrated model
	Obs	a set of observables
	P	one property
	Ctx	property context (optional)
	Res	path restrictions (optional)
Outputs	R	A flag that indicates whether P is diagnosable (YES) or not (NO) or (UNKNOWN) in case no counterexample has been found using BMC in hybrid models.
	TrA	A trace exhibiting property P.
	TrB	A trace NOT exhibiting P, but whose observational data is identical to TrA.
Using model checking techniques, the COMPASS toolset checks if an operational model, extended with a fault model, is diagnosable with respect to a given diagnosability property. A property is said to be diagnosable if the observational data is always sufficient for the controller to infer whether the property is given at a specific moment in time, or not. This analysis can be restricted to situations where the FSM has in past exhibited a given ordered set of path restrictions, one at a time. The user may also specify a context (always a propositional formula) in which the diagnosability property is to be considered.		

```
$> scripts/check_diagnosability.py \
    -p <properties_file.propxml> \
    [--property-num=N] \
    [--path-restrictions <properties_file.propxml>] \
    [--property-context <property_file.propxml>] \
    [--bmc-length <bound>] \
    [--diag-delay <delay>] \
    file1.slim ... fileN.slim
```

A property is a propositional formula. So are the path restrictions, where one property corresponds to one state restriction. For both files the format is the XML property file format. As to the path restrictions, the order in which they are specified in the file is the chronological order in which they are applied to the execution paths of the FSM. Option –**property-num** allows to specify which properties exactly is to be considered as a fault. Option **--diag-delay** and option **--bmc-length** specify that the property must be diagnosable within a SAT Bound and a specific time bound (maximum diagnosability bound). Note that currently the set of observables is taken directly from the input SLIM models, and this set has to be non-empty. An example for a diagnosable property that holds:

```
$> scripts/check_diagnosability.py -i \
    -p documentation/examples/battery_sensor/system_fdir.propxml \
    --property B1_LOW \
```

```
    documentation/examples/battery_sensor/system_fdir.slim
...
A diagnosability check has been performed for the property \
   (sys.psu1.battery.low).
No counterexample for diagnosability of the given property \
   was found within the given bound.
```

On the other side, a different property doesn't hold:

```
$> scripts/check_diagnosability.py -i \
    -p documentation/examples/battery_sensor/system_fdir.propxml \
    --property S1_FAULT \
    documentation/examples/battery_sensor/system_fdir.slim
...
A diagnosability check has been performed for the property \
  (sys.sensor1.error=error:dead).
The property was found to be not diagnosable (see counterexample \
   in system_fdir/system_fdir_trace1_A.xml and \
   system_fdir/system_fdir_trace1_B.xml).
```

A.1.8 Fault Detection Analysis

Check whether a fault is detectable in a model		
Inputs	M	an operational model
	FM	a fault model
	FDM	an FDIR model (optional)
	Alm	a set of alarms
	F	a fault
Outputs	O	a set of candidate detection means (a set of alarms).
Using model checking techniques, the COMPASS toolset checks if a fault is detectable in an operational model, extended with a fault model and optionally extended with an FDIR model. A fault is considered detectable if there exists an alarm in the input set of alarms, such that every occurrence of the fault eventually causes the alarm to be true.		

```
$> scripts/compute_fault_detection.py -i \
    -p <properties_file.xml> \
    [--property-num=N] \
    [--bmc-length] \
    [--detection-delay] \
    file1.slim ... fileN.slim
```

Currently, the script allows any propositional formula to be considered as a fault. Thus a fault can be specified in a properties file which is allowed to have only purely propositional properties. Option –**property-num** allows to specify which properties exactly is to be considered as a fault. Option **--detection-delay** and option **--bmc-length** specify that the

property must be detectable within a SAT Bound and a specific time bound (maximum detection bound). Note that currently the set of alarms is taken directly from the input SLIM models, and this set has to be non-empty. The output set of alarms is generated into file with extension `.flt_detect`. If a fault is not detectable the returned set is empty. For example:

```
$> scripts/compute_fault_detection.py \
    -i -p documentation/examples/adder/adder_discrete/adder.propxml \
    --property-num 3 \
    documentation/examples/adder/adder_discrete/adder.slim \
    documentation/examples/adder/adder_discrete/adder_err.slim
```

generates file `adder/adder.flt_detect` with the set of alarms.

A.1.9 Fault Isolation Analysis

Compute fault isolation measures for a model		
Inputs	M FM FDM Alm	an operational model a fault model an FDIR model (optional) a set of alarms
Outputs	set of FT	a set of fault trees, one for each alarm.
Using model checking techniques, the COMPASS toolset computes fault isolation measures for an operational model, extended with a fault model and an optional FDIR model. Fault isolation measures are produced by computing for each input alarm the set of minimal explanations (single faults or combinations thereof) that are compatible with the alarm being true. The set of minimal explanations is presented as a set of fault trees, one for each alarm.		

```
$> scripts/compute_fault_isolation.py -i \
    file1.slim ... fileN.slim
```

Note that the set of alarms is taken directly from the input SLIM models, and this set has to be non-empty. The fault trees are generated as pairs of files: `.N.flt_events` for events and `.N.flt_gates` for gates. Here `N` is an integer number ranging from 0 to the number of alarms. For example:

```
$> scripts/compute_fault_isolation.py -i \
    documentation/examples/adder/adder_discrete/adder.slim \
    documentation/examples/adder/adder_discrete/adder_err.slim
```

generates files `adder/adder.0.flt_events`, `adder/adder.0.flt_gates`, `adder/adder.1.flt_events`, `adder/adder.1.flt_gates`, `adder/adder.2.flt_events` and `adder/adder.2.flt_gates`, i.e. three fault trees for alarms `bit1_alarm1`, `adder_alarm1` and `bit2_alarm1`, respectively. It is possible to see this information by running the fault tree viewer, e.g.:

```
$> scripts/view_ft.py \
    --events-file adder/adder.0.flt_events \
    --gates-file adder/adder.0.flt_gates
```

A.1.10 Fault Recovery Analysis

Check the capability of a system to recover from a fault.		
Inputs	M FM FDM P	an operational model a fault model an FDIR model (optional) a property
Outputs	$true\|false$ Tr	*true* if recovery property holds, and $false$ otherwise, counterexample trace if property does not hold.
Using model checking techniques, the COMPASS toolset checks if a system is able to recover from a fault, given an operational model, extended with a fault model and an optional FDIR model, and a property representing the recoverability. A counterexample in XML format shall be generated if the property does not hold.		

```
$> scripts/compute_fault_recovery.py \
    [--bmc [--bmc-length=<length>]] \
    [-i] -p <properties_file.xml> \
    file1.slim ... fileN.slim
```

This script is run exactly the same and with the same options as `check_properties.py`. See Section A.1.2 for more details. An example of running the script is:

```
$> scripts/compute_fault_recovery.py -i \
    -p documentation/examples/adder/adder_discrete/adder.propxml \
    documentation/examples/adder/adder_discrete/adder.slim \
    documentation/examples/adder/adder_discrete/adder_err.slim
```

A.1.11 Fault Coverage Analysis

Categorize faults depending on whether they can be detected/recovered from, into different classes.		
Inputs	M FM FDM FP P F	an operational model a fault model an FDIR model (optional) a property xml file a property a fault
Outputs	$true\|false$	*true* if coverage property holds, and $false$ otherwise.
Using model checking techniques, the COMPASS toolset checks if a system is able to categorize faults depending on whether they can be detected/recovered from, into different classes.		

```
$> scripts/compute_fault_coverage.py [-i] \
    [--bmc-length=<length>] \
    [-p <properties_file.xml>] \
```

```
[--property-num=N] \
[--fault-str="fault_name"] \
[--detection-delay=<length>] \
file1.slim ... fileN.slim
```

The script performs the analysis on all the valid properties found in the specified properties file. Option –**property-num** allows to specify the exact propery to be used, whil option –**fault-str** specifies the fault to be detected. Option **--bmc-length** and option **--detection-delay** have the same behavior shown in Section A.1.8.

Results are printed out to stdout, and possibly generated traces are dumped into XML files (one file each trace).

For example:

```
$> scripts/compute_fault_coverage.py -i \
    -p documentation/examples/adder/adder_discrete/adder.propxml \
    --property-num=3 \
    --fault-str="bit1.error = error:inverted" \
    --detection-delay=-1.0 \
    --bmc-length=50 \
    documentation/examples/adder/adder_discrete/adder.slim \
    documentation/examples/adder/adder_discrete/adder_err.slim

...
Fault Coverage Analysis has been performed for fault
bit1.error = error:inverted  and the result is: Covered
```

A.1.12 Zeno Detection

Check if a mode of the model is Zeno.		
Inputs	M FM FDM	an operational model a fault model (optional) an FDIR model (optional)
Outputs	$Ok\|Unknown$ Tr	Ok if Zeno cycle is not detected, $Unknown$ otherwise, counterexample trace if Zeno cycle is detected.
Using model checking techniques, the COMPASS toolset checks if a system is able to detect from a mode, given an operational model, extended with a fault model and an optional FDIR model, and optional bounds. A counterexample in XML format shall be generated if the Zeno cycle is detected.		

```
$> scripts/check_zeno_states.py [-i] \
    [--bmc-length=<length>] \
    [--klive-bound=<length>] \
    file1.slim ... fileN.slim
```

Depending on the engine we want to use when performing zenoness checking (BMC or kliveness) a different bound can be specified using, respectively, option **--bmc-length** and option **--klive-bound**.

Results are printed out to stdout, and possibly generated traces are dumped into XML files (one file each trace).

For example:

```
$> scripts/check_zeno_states.py /
    --bmc-length=5 /
    --klive-bound=10 /
    documentation/examples/features/zeno/zeno_01.slim

...
Zenoness Checking: In "subsys1" checking of:
"(root.sc_subsys1.mode = mode_l2)" is UNREACHABLE

Zenoness Checking: In "(root)" checking of:
"(root.mode = mode_DefaultInitialMode)" is NON-ZENO
(trace in 'zeno_01/zeno_01_trace2.xml')
```

In the example, the first Zeno cycle has been found to be UNREACHABLE. In contrast, the second Zeno cycle has been found to be NON-ZENO and a corresponding counter-example has been generated in `zeno_01/zeno_01_trace2.xml`.

To see the content of a trace, the **trace viewer** can be used:

```
$> scripts/view_trace.py <trace-file.xml>
```

A.1.13 Time Divergence Detection

Detect time divergent clocks in the model.		
Inputs	M	an operational model
	FM	a fault model (optional)
	FDM	an FDIR model (optional)
	CLB	a CLOCK Bound (optional)
	FCLB	a CLOCK Bound CVS file (optional)
Outputs	*Ok*|*Unknown*	*Ok* if time divergent path is not detected, *Unknown* otherwise,
	Tr	counterexample trace if time divergent path is detected.
Using model checking techniques, the COMPASS toolset checks if a system is able to detect from a clock, given an operational model, extended with a fault model and an optional FDIR model, and optional bounds. A counterexample in XML format shall be generated if the time divergent path is detected.		

```
$> scripts/check_time_divergence.py [-i] \
    [--bmc-length=<length>] \
    [--clock-bound=<length>] \
    [--gen-clock-bound] \
    [--clock-bound-file=<file.cvs>] \
    file1.slim ... fileN.slim
```

Option **--clock-bound** is used to specify the clock max bound used when performing clock divergence checking (it verifies whether the clock is *bounded* or not with respect to this value); this value is ignored when reading from clock bound file, which is a CSV file which can be specified using option **--clock-bound-file** or can be created through option **--gen-clock-bound**. In this case, an entry is created for each clock data subcomponent, associated to a default clock bound which is, if specified, the value of option **--clock-bound**.

Results are printed out to stdout, and possibly generated traces are dumped into XML files (one file each trace).

For example:

```
$> scripts/check_time_divergence.py \
    --clock-bound-file=documentation/examples/features/clock/unbound_input.cvs \
    --bmc-length=20 \
    documentation/examples/features/clock/clock.slim

Clock-Divergence Checking in "subsys1":
    clock "x" set with bound "2" is UNBOUNDED (trace in
'clock_01/clock_01_trace1.xml')
```

In the example, the clock has been found to be UNBOUNDED, and a corresponding counter-example has been generated in `clock_01/clock_01_trace1.xml`.

Another example could be:

```
$> scripts/check_time_divergence.py \
    --clock-bound-file=documentation/examples/features/clock/unbound_input.cvs \
    --bmc-length=1 \
    documentation/examples/features/clock/clock.slim

Clock-Divergence Checking in "subsys1":
    clock "x" set with bound "2" is UNKNOWN
```

In the example, the clock has been found to be UNKNOWN and no counter-example has been generated.

A.1.14 Performability Evaluation

For analysing the system on nominal performance (performance evaluation), reliability, availability and maintainability (dependability) and performance under degraded conditions (performability), the toolset shall verify whether an operational model extended with a fault model, optionally extended with an FDIR model, satisfies a probabilistic property.		
Inputs	M FM FDM P	an operational model a fault model (optional) an FDIR model (optional) a set of probabilistic properties
Outputs	[0,1]	probability for each probabilistic property $\phi \in P$
Using numerical probabilistic model checking techniques, the toolset shall verify whether an operational model extended with a fault model, optionally extended with a FDIR model, satisfies the given probabilistic property.		

```
$> scripts/evaluate_performability.py -i \
    file1.slim ... fileN.slim
$
```

An example of the running script is:

```
$> scripts/evaluate_performability.py -i \
    documentation/examples/sensorfilter/sensorfilter.slim \
    documentation/examples/sensorfilter/sensorfilterErr.slim \
    -p documentation/examples/sensorfilter/sensorfilter.propxml \
    --property="sensor1 dead in [1,50]"
    --error-bound=1e4
Error bound possibly too high
The root component contains ports, performability assumes a closed world.
Performability Evaluation of 'Globally, it is always the case that
{sensors.sensor2.error = error:OK} holds between 0 and 1 with
probability > 0' returns
prob = 0.0062508029
$
```

The performability analysis is performed for each probabilistic property expressed in the model and in the order they are expressed. For each probabilistic property, the corresponding probability range is outputted.

A.1.15 Fault Tolerance Evaluation

Compute fault tolerance measures given a set of properties and a corresponding set of fault trees.		
Inputs	P FTs	a set of properties a set of fault trees
Outputs	FTM	the fault tolerance measure. property $\phi \in P$
Given a component, a set of properties, and a set of corresponding fault trees generated for the component, the COMPASS toolset shall compute fault tolerance measures. Fault tolerance measures shall reect the capability of the component to sustain single or multiple faults and shall be computed as follows: for each cardinality, a table shall report the number of unique minimal cut sets with that cardinality.		

```
$> scripts/evaluate_fault_tolerance.py \
    --fts-files <fault_tree_file_1.xml> \
    --fts-files <fault_tree_file_2.xml> \
    ...
    file1.slim ... fileN.slim
$
```

An example of the running script is:

```
$> scripts/evaluate_fault_tolerance.py \
    --fts-file documentation/examples/adder/Adder_flt_ftoe_1.xml \
    --fts-file documentation/examples/adder/Adder_flt_ftoe_2.xml \
    documentation/examples/adder/adder_discrete/adder.slim \
    documentation/examples/adder/adder_discrete/adder_err.slim

Fault Tree Evaluation results are stored in "adder/adder.ftoe".
$
```

The analysis counts the number of unique cut sets computed for all the given fault trees (specified using the option **fts-file**), divided on the basis of their cardinality.

A.1.16 Dynamic Fault Tree (and Criticality) Evaluation

Compute the probabilities of the events in a fault tree and the criticality number.		
Inputs	FT TB	a dynamic fault tree upper time bound
Outputs	FTO	a fault tree with probabilities attached to events.
By translating the fault tree to its corresponding Markov Chain, the toolset shall use probabilistic model checking techniques to compute the probabilities of occurrence of the top level event and the intermediate events in the fault tree within the given time bounds. The output fault tree shall be generated in XML or other suitable format[1]. Additionally, the toolset shall compute the criticality number associated with the top level event, as the product between a user-defined severity number of the property violation (associated with the top level event) and the probability of the top level event..		

```
$> scripts/evaluate_dynamic_fault_tree.py \
    <fault_tree_xml> \
    <rates_map_file> \
    <upper_bound> \
    <output_fault_tree_events_filename> \
    <output_fault_tree_gates_filename> \
    --severity=<severity> \
```

This script requires to specify a fault tree in xml format, a rates-map file (note that it is generated during fault tree generation) an upper bound for the mission time and the name of the events and gates files of the resulting annotated fault tree.

Both the fault tree (in XML) as well as the rates map (mapping error rates to basic events) can be obtained by running the `compute_fault_tree.py` script. The mission time defines the time bound for which the failure probability of the fault tree is calculated.

The following is an example of running this cli-script:

```
$> scripts/evaluate_dynamic_fault_tree.py \
    documentation/examples/sensorfilter/sensorfilter_ft.xml \
    documentation/examples/sensorfilter/sensorfilter_ft_rates.rates_map \
    100 \
    annotated-ft.events \
    annotated-ft.gates \
    --severity=5
The fault tree has been evaluated. The annotated fault tree can be found
in the files sensorfilter_ft/annotated-ft.events and
sensorfilter_ft/annotated-ft.gates.
$
```

The resulting fault tree can be viewed with the following command:

```
$> scripts/view_ft.py \
    --events-file sensorfilter_ft/annotated-ft.events \
    --gates-file sensorfilter_ft/annotated-ft.gates
```

A.1.17 Dynamic Fault Tree Verification

Verify whether a probabilistic property holds for a dynamic fault tree.		
Inputs	FT P	a dynamic fault tree a probabilistic property
Outputs	Prob	The probability for each probabilistic property.
By translating the dynamic fault tree to its corresponding Markov Chain, the toolset shall use probabilistic model checking techniques to verify whether the probabilistic property holds for the dynamic fault tree.		

```
$> scripts/verify_dynamic_fault_tree.py \
    <fault_tree_xml> \
    <rates_map_file> \
    -p <property_file>
```

Note that the rates-map file is generated during fault tree generation. The property file can state multiple probabilistic properties. The cli-script then computes the probability for each of them.

The following is an example of running this cli-script:

```
$> scripts/verify_dynamic_fault_tree.py \
    documentation/examples/sensorfilter/sensorfilter_ft.xml \
    documentation/examples/sensorfilter/sensorfilter_ft_rates.rates_map \
    -p documentation/examples/sensorfilter/sensorfilter_ft_properties.xml
Fault tree verification for 'Globally, {Top_Level_Event} holds
eventually between 10 and 50 with probability > 0' returns
prob = 0.784841
Fault tree verification for 'Globally, {Top_Level_Event and not E6}
holds eventually between 0 and 50 with probability > 0' returns
prob = 0.0
Fault tree verification for 'Globally, {E2} holds without interruption
until {fault_cfg_4} holds between 20 and 100 with probability > 0'
returns
prob = 0.0
Fault tree verification for 'Globally, it is always the case that
{fault_cfg_2} holds between 0 and 100 with probability > 0' returns
prob = 0.0
Fault tree verification for 'Globally, if {fault_cfg_3} holds then it
must be the case that {fault_cfg_4} has occurred before between 0 and
100 with probability > 0' returns
prob = 1.0
Fault tree verification for 'Globally, if {E5} has occurred then in
response {Top_Level_Event} eventually holds between 0 and 10 with
probability > 0' returns
prob = 2.30000000001e-06
$
```

A.1.18 Monte Carlo Simulation

Analyze the probability of a probabilistic property by means of Monte-Carlo simulation		
Inputs	M FM FDM P	an operational model a fault model (optional) an FDIR model (optional) a set of probabilistic properties
Outputs	[0,1]	probability for each supported probabilistic property $\phi \in P$
Using statistical analysis techniques (Monte-Carlo simulation), the toolset determines the probability that the specified property holds, givens the extended input specification. The confidence and error bound parameters control the accuracy and precision of the simulation.		

```
$> scripts/simulate_with_monte_carlo.py [-i]\
    -p <properties_file.xml>
    [--confidence <confidence>]\
    [--error-bound <error bound>]
    file1.slim ... fileN.slim \
$
```

The properties file contains the properties for which the probability will be determined by simulation. The confidence and error bound parameters are optional. The confidence must be specified as a value between 0.0 and 1.0. The error bound is any (small) positive value.

An example of the running script is:

```
$> scripts/simulate_with_monte_carlo.py -i \
    -p documentation/examples/gps/gps_prop_prob.propxml \
    documentation/examples/gps/gps.slim
Monte Carlo Simulation of 'Globally, {signal = 0} holds eventually between
0 and 5 with probability > 0' returns
(0.0, 0.05)
```

The Monte-Carlo simulation is performed for each supported probabilistic property expressed in `gps_prop_prob.xml` and in the order they are expressed. For each probabilistic property, the corresponding probability range is outputted.

A.1.19 Validation of Formal Properties

Check different types of properties given by command line. In more details, check consistency of a set of properties; check wheather a set of properties is consistent with an scenario, specified with a new formal property. Finally, check if a set of properties entail an assertion, specified with a new formal property.		
Inputs	M	SLIM model
	type of check	consistency, possibility, or assertion
	component	component name or 'ALL' to consider ALL components
	r	a subset of properties (contracts ids) separated by comma, or ALL to consider ALL contracts and subcomponents contracts.
	p	assertion/possibility (optional)
	u	compute unsat cores (optional)
	w	consider the whole architecture for the check (optional)
Outputs	OK	A trace is shown as witness for consistent and possibility check, or unsat cores for an assertion check if 'u' option is given
	NOT OK	A trace is given as counterexample for an assertion check, or unsat cores are given for consistency and possibility if 'u' option is given.
The input slim model is translated to an ocra specification and then OCRA tool is used as backend to check the validity of the given property.		

```
$> scripts/check_validation_properties.py \
    [[--consistency] | [--possibility] | [--assertion]] \
    [--component=<component_name>] \
    [--r=<subset_prop>] \
    [--p=<ocra_prop] \
    [--u] \
    file.slim
```

The type of validation check is chosen with option **`--consistency`**, option **`--possibility`** or option **`--assertion`**. Option **`--component`** specifies the component, while option **`--r`** is used to define the subset of properties we are considering (`ALL` to include all contracts and subcomponent contract). Option **`--p`** can be used only with option **`--possibility`** and option **`--assertion`** and it represent the contract id on which we want to run the check. Option **`--u`** computes the unsat cores.

The following is an example of running the cli-script:

```
$> scripts/check_validation_properties.py \
    --assertion \
    --component="Sys" \
    --r="Sys.termination.ASSUMPTION,E.cmd_fw.GUARANTEE" \
    --p="Sys.termination.GUARANTEE" \
    --u \
    documentation/examples/starlight/starlight.slim
```

```
...
Entailment Input_validation_prop: UNKNOWN
...
```

Formulas ids are used to make reference to assumption, guarantee, and norm guarantee (i.e., $A \rightarrow G$, where A is the assumption and G is the guarantee of a given contract) of the contracts of components. A formula id (e.g., Sys.termination.GUARANTEE) is made up of a component name, a contract name and one of the keyword: *ASSUMPTION*, *GUARANTEE*, or *NORM_GUARANTEE*.

A.1.20 Tighten a Contract Refinement

Tighten a given contract refinement by weaking the assumption of the parent contract and the guarantee of its subcontracts (top-down approach) or strengthening the guarantee of the parent contract and the assumption of its subcontracts (bottom-up approach).		
Inputs	M tightening approach contract component s	SLIM model top-down or bottom-up contract name component name which is defined the given contract consider other contracts refinement which have subcontracs in common (optional)
Outputs	List	List of tighten contracts
The input slim model is translated to an ocra specification and then OCRA tool is used as backend to tighten the given contract refinement.		

```
$> scripts/tighten_contract_refinement.py \
    [[--top_down] | [--bottom-up]] \
    [--component=<component_name>] \
    [--contract=<contract>] \
    [--s] \
    file.slim
```

This script requires a component and a contract specified through option **`--component`** and option **`--contract`**. The type of tightening can be chose using option **`--top-down`** or option **`bottom-up`**. Option **`--s`** allows to take into account those contracts refinements at same level which have subcontracts in common.

The following is an example of running the cli-script:

```
$> scripts/tighten_contract_refinement.py \
    --top_down \
    --component="Sys" \
    --contract="termination" \
    documentation/examples/starlight/starlight.slim
...
-Tighten Result N '1' for Weakening Assumption of Parent Contract 'termination'
TRUE
...
```

A.1.21 Check Contracts Composite Implementation

Check compositionally if a component implementation satisfies a contract (or all contracts).		
Inputs	M contract	SLIM model contract name
Outputs	Val	The result of the composite verification; it can be `OK`, `BOUND OK` or `NOT OK`
The input slim model is translated to an ocra specification and then OCRA tool is used as backend to check compositionally the contract(s). The verification is compositional in the sense that if the component implementation is composite, the correctness of the component is verified by checking the correctness of the contract refinement and the correctness of the atomic components with regards to their contracts.		

```
$> scripts/check_contracts_composite_implementation.py \
  [[--bmc] | [--klive] | [--kzeno] \
  [--fairness] \
  [--contract <contract>] \
  [--bound <ocra_check_bound>] \
  file.slim
```

This script requires to specify the contract for which the composite verification will be run through option **`--contract`**. The engine to be used can be chosen using option **`--bmc`**, option **`--klive`** or option **`--kzeno`**. Option **`--fairness`** enables the addition of fairness assumption.

The following is an example of running the cli-script:

```
$> scripts/check_contracts_composite_implementation.py \
    --kzeno --fairness --bound=5 \
    --contract=termination \
    documentation/examples/starlight/starlight.slim
...
Checking refinement of component: Sys
  Checking "CONTRACT termination REFINEDBY sc_E.cmd_fw, sc_Low.rx, sc_Low.causality,
  sc_High.rx, sc_High.causality, sc_E.res_fw;"
    Checking the correct implementation of "termination" ... [OK]
    Checking the correct environment of "sc_E.cmd_fw" .. [OK]
    Checking the correct environment of "sc_Low.rx" .. [OK]
    Checking the correct environment of "sc_Low.causality" .. [OK]
    Checking the correct environment of "sc_High.rx" .. [OK]
    Checking the correct environment of "sc_High.causality" .. [OK]
    Checking the correct environment of "sc_E.res_fw" .. [OK]
...
```

A.1.22 Check Contracts Monolithic Implementation

Check monolithically if a component implementation satisfies a contract (or all contracts).		
Inputs	M contract	SLIM model contract name
Outputs	Val	The result of the monolithic verification; it can be `OK`, `BOUND OK` or `NOT OK`
The input slim model is translated to an ocra specification and then OCRA tool is used as backend to check compositionally the contract(s). The verification is monolithic in the sense that OCRA generates an SMV file representing the behavior of the whole component and verifies it with one check.		

```
$> scripts/check_contracts_monolithic_implementation.py \
  [[--bmc] | [--klive] | [--kzeno] \
  [--fairness] \
  [--contract <contract>] \
  [--bound <ocra_check_bound>] \
  file.slim
```

This script requires to specify the contract for which the monolithic verification will be run through option **`--contract`**. The engine to be used can be chosen using option **`--bmc`**, option **`--klive`** or option **`--kzeno`**. Option **`--fairness`** enables the addition of fairness assumption.

The following is an example of running the cli-script:

```
$> scripts/check_contracts_monolithic_implementation.py \
    --kzeno --fairness --bound=5 \
    --contract=termination \
    documentation/examples/starlight/starlight.slim
...
Checking implementation of contract Sys.termination ... [OK]
...
```

A.1.23 Check Contracts Refinements

Check if the refinement of a contract (or all contracts) is correct		
Inputs	M contract	SLIM model contract name
Outputs	Msg	A message saying whether all contracts refinements are ok are not
The input slim model is translated to an ocra specification and then OCRA tool is used as backend to check the contract(s) refinement.		

```
$> scripts/check_contracts_refinement.py \
  [[--bmc] | [--klive] | [--kzeno] \
  [--fairness] \
  [--contract <contract>] \
```

```
  [--bound <ocra_check_bound>] \
  file.slim
```

This script requires to specify the contract for which the refinement check will be run through option **--contract**. The engine to be used can be chosen using option **--bmc**, option **--klive** or option **--kzeno**. Option **--fairness** enables the addition of fairness assumption.

The following is an example of running the cli-script:

```
$> scripts/check_contracts_refinement.py \
    --klive --fairness --bound=5 \
    --contract=termination \
   documentation/examples/starlight/starlight.slim
...
Summary: everything is OK
...
```

A.1.24 Generate Hierarchical Fault Tree

Generate a hierarchical fault tree from the contract specification		
Inputs	M contract	SLIM model contract name
Outputs	Msg	A message saying whether all contracts refinements are ok are not
The input slim model is translated to an ocra specification and then OCRA tool is used as backend to generate the fault tree.		

```
$> scripts/compute_fault_tree_from_contracts.py \
  [[--bmc] | [--klive] \
  [--fairness] \
  [--contract <contract>] \
  [--bound <ocra_check_bound>] \
  [--view] \
  file.slim
```

This script requires to specify the contract for which we want to run hierarchcal fault tree generation through option **--contract**. The engine to be used can be chosen using option **--bmc** or option **--klive**. Option **--fairness** enables the addition of fairness assumption. With option **--view** it is possible to visualize the fault tree once it has been generated.

The following is an example of running the cli-script:

```
$> scripts/compute_fault_tree_from_contracts.py \
    --klive --fairness --bound=5 \
    --contract=termination \
   documentation/examples/starlight/starlight.slim
...
FT+ generated: Sys.termination-FAILURE_0
...
```

The generated fault tree can now be visualized using the following command:

```
$> scripts/view_ft.py \
   --events-file starlight/Sys.termination-FAILURE_0-events.txt \
   --gates-file starlight/Sys.termination-FAILURE_0-gates.txt
```

We can both generate the fault tree and visualize it using only one command as follows:

```
$> scripts/compute_fault_tree_from_contracts.py \
    --klive --fairness --bound=5 --contract=termination --view \
    documentation/examples/starlight/starlight.slim
```

A.1.25 TFPG Syntax Check

Check if a TFPG is syntactally correct		
Inputs	T	TFPG model
Outputs	m	A message saying whether all checks passed or not
Several syntactic checks are run on a particular TFPG.		

The following is an example of running the cli-script:

```
$> scripts/check_syntax_tfpg.py \
    --tfpg-file documentation/examples/battery_sensor/system.txml

Validating TFPG file 'documentation/examples/battery_sensor/system.txml'...
All checks passed.
```

A.1.26 TFPG Behavioral Validation

Run TFPG behavioral validation.		
Inputs	M	an operational model
	A	a SLIM Association xml file
	FM	a fault model
	FDM	an FDIR model (optional)
	TFPG	a TFPG xml file
Outputs	$Ok \vert Unknown$	*Ok* if is behaviourally correct, *Unknown* otherwise,
	TrA	counterexample TFPG trace if it is not complete.
	TrB	counterexample SYSTEM trace if it is not complete.
It is used to automatically validate the behavioral of a TFPG, starting from a system model and a definition of the SLIM associations elements: a set of failure modes, a set of monitored discrepancies, a set of non-monitored discrepancies, and a set of system modes.		

```
$> scripts/validate_tfpg_behavior.py \
    [-i] \
    [--bmc-length=<length>] \
    [--tfpg-file <file_tfpg.txml>] \
    [--associations-file <file_associations.axml>] \
    file1.slim ... fileN.slim
```

Results are printed out to stdout, and possibly generated traces are dumped into XML files (one file each trace).

For example:

```
$> scripts/validate_tfpg_behavior.py \
    -i \
    --tfpg-file documentation/examples/battery_sensor/system.txml \
    --associations-file documentation/examples/battery_sensor/system.axml \
    --bmc-length 10
    documentation/examples/battery_sensor/system.slim \

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

In the example, all checks passed and the result is bounded_completeness. So no traces are generated.

Another example could be:

```
$> scripts/validate_tfpg_behavior.py \
    -i \
    --tfpg-file documentation/examples/battery_sensor/system_incomplete.txml \
    --associations-file documentation/examples/battery_sensor/system.axml \
    --bmc-length 10
    documentation/examples/battery_sensor/system.slim

...
Result: The TFPG is incomplete with respect to the model.
A pair of traces have been generated.
```

In the example, all checks passed and the result is incomplete, traces have been generated in `system/system_trace1-TFPG.xml` and `system/system_trace1-SYS.xml`.

A.1.27 TFPG Synthesis

Run TFPG synthesis.		
Inputs	M A FM FDM	an operational model a SLIM Association xml file a fault model an FDIR model (optional)
Outputs	*Ok*|*Unknown* TFPG	*Ok* if the checks passed, *Unknown* otherwise, the synthesized TFPG xml file generated.
It is used to automatically synthesize a TFPG graph, starting from a system model and a definition of the SLIM associations elements: a set of failure modes, a set of monitored discrepancies, a set of non-monitored discrepancies, and a set of system modes.		

```
$> scripts/tfpg_synthesis__srs_097.py \
    [-i] \
    [--bmc-length=<length>] \
    [--associations-file <file_associations.axml>] \
    file1.slim ... fileN.slim
```

Results are printed out to stdout, and possibly generated traces are dumped into XML files (one file each trace).

For example:

```
$> scripts/synthesize_tfpg.py \
    -i \
    --bmc-length=20 \
    --associations-file=documentation/examples/battery_sensor/system.axml \
    documentation/examples/battery_sensor/system.slim


...
Result: ok
  tfpg file: <path_to_top_dir>/system_synth_tfpg.txml
```

In the example, all checks passed and a TFPG is synthesized and has been generated in `system_synth_tfpg.txml`. The created tfpg can be visualized using the corresponding cli-script:

```
$> scripts/view_tfpg.py \
    --tfpg-file system_synth_tfpg.txml
```

A.1.28 TFPG Effectiveness Validation

Run TFPG effectiveness validation.		
Inputs	TFPG FM SM	a TFPG xml file a set of Failure Modes the System Mode (optional)
Outputs	*Ok*|*Unknown* TrA TrB	*Ok* if the TFPG is effective, *Unknown* otherwise, counterexample trace if a FM of SM is not diagnosable. counterexample trace if a FM of SM is not diagnosable.
It is used to know if the TFPG is effective for diagnosis, starting from the TFPG and a set of target modes: failure modes.		

```
$> scripts/validate_tfpg_effectiveness.py \
    [--tfpg-file <file_tfpg.txml>] \
    [--use-bmc=<boolean>]
    [--bmc-length=<length>] \
    [--target-fm-set=<FM1,...,FMn>] \
    [--sampling-rate=<N>] \
    [--target-system-mode=SM1,...,SMn>]
```

Option **target-fm-set** specifies the set of FMs that need to be diagnosed (as a group), option **sampling-rate** represents the interval at which the monitored system is sampled and option **target-system-mode** specifies the system mode for which effectiveness should be analyzed.

Results are printed out to stdout, and possibly generated traces are dumped into XML files (one file each trace).

For example:

```
$> scripts/validate_tfpg_effectiveness.py \
    --tfpg-file documentation/examples/battery_sensor/system.txml \
    --target-fm-set=Gen1_off \
    --sampling-rate=1.0 \
    --target-system-mode=Primary \
    --use-bmc \
    --bmc-length=10


...
No counterexample has been found within bound 10.
```

In the example, all checks passed and no counterexample has been found within bound 10.

Another example could be:

```
$> scripts/validate_tfpg_effectiveness.py \
    --tfpg-file documentation/examples/battery_sensor/system.txml \
    --target-fm-set=Gen1_off \
    --sampling-rate=1.0 \
    --target-system-mode=Secondary2 \
    --use-bmc \
    --bmc-length=10


...
The TFPG is not effective for diagnosis under the given constraints
(see counterexample in documentation/examples/battery_sensor/system_trace1-A.xml and
documentation/examples/battery_sensor/system_trace1-B.xml).
```

In the example, all checks passed and the result is that the TFPG is not effective under the given constraints. So traces have been generated in `battery_sensor/system_trace1-A.xml` and `battery_sensor/system_trace1-B.xml`.

A.1.29 Advanced Script Options

This section describes the script options which are not required by default. The description of all of them can be obtained by running any script with the option **`--help`** option, e.g.:

```
$> scripts/check_syntax.py --help
```

In general, options have two forms: the short one consisting of option **-** and one character, and the long one consisting of option **--** and a hyphen-separated words. Options may have either of the form or both. For example, option **-h** and option **--help** are completely equivalent.

Here is the description of the most common options:

- option **--version** – outputs the version of the script
- option **-h**, option **--help** – outputs the help message
- option **-f**, option **--file** – specifies base-name for all generated files. By default the base-name is the name of the first input SLIM file without extension.
- option **-d**, option **--dir** – allows to specify the directory where the files are to be generated. This path is merged with base-name to obtain the full path.
- option **-r**, option **--root** – specifies the name of the root component implementation. This options is required if several component implementations can be root and the script cannot detect which one to use.
- option **-l**, option **--level** – sets the output system logging level. Note that this option is mainly designed for debugging the toolset.
- option **-e**, option **--logfile** – sets the name of the log file where to dump all the output messages.
- option **--quiet** – only error message are output to the console. Nominal messages are not output.
- option **--stop-after** – allows to stop the script after a particular phase.
- option **-i**, option **--injection** – specifies that fault injections are to be applied.
- option **--output-fault-extended** – makes the script to output `.err_slim` file with input model after fault extension is done. This is purely a debugging option.
- option **--property** – Specify the name of the property to analyze. This option can be given multiple times.
- option **--property-instance** – Specify the path of the instance of which to check its specified properties.
- option **-p**, option **--property-file** – specifies a properties file. Use the option if the properties are specified in a separate file, as opposed to specified directly in the model.
- option **--smv_int_to_word** – by default all the data of type `int` are treated as unbounded integers. This may easily cause a state space explosion and is not supported by all types of analysis. Using this option will encode integers as bounded words.

- option **-w**, option **--wordsize** – specifies the number of bits to encode integers with. For example option **-w 3** makes integers be encoded as 3 bits, which is enough to deal with number from 0 to 7.

Arguments of some options, such as option **--level**, range over particular sets of strings. To see which string exactly are allowed a user can run a script with this options and some invalid argument and the script will output the allowed set in the error message.

In case of TFPG analyses, the following options are common:

- option **--tfpg-file** – specifies the name of the file containing the representation of the tfpg (which can be in xml format or in a more readable format).
- option **--associations-file** –specifies the name of the file containing the tfpg slim associations, wich is a file containing a list of slim expressions, each one linked to an element of the tfpg.