COMPASS

Correctness, Modeling, and Performance of Aerospace Systems

COMPASS Tutorial

Version 3.0.1

Prepared by
Fondazione Bruno Kessler
RWTH Aachen University

Contents

Chapter 1

Introduction

This tutorial introduces the main features of the COMPASS toolset – it covers both modeling and verification. It is intended as a hands-on document that can be used as a starting point to quickly learn how to develop and analyze models in COMPASS. This tutorial does not cover all the functionalities and options of the COMPASS toolset. It is complemented by the COMPASS User Manual [3], that can be consulted for a more systematic reference to the COMPASS toolset.

This document is structured as follows.

- Chapter 2 lists the (abbreviated) terms that are applied in this document.
- Chapter 3 presents an overview of COMPASS and the COMPASS workflow.
- Chapter 4 introduces the SLIM language, and discusses how to write properties and perform the basic simulation and verification analyses.
- Chapter 5 deals with the specification of faults and errors, fault propagation, and introduces model-based safety assessment.
- Chapter 6 discusses how to write models with probabilities, and perform probabilistic analyses and safety assessment.
- Chapter 7 introduces Fault Detection, Identification and Recovery (FDIR) and the related notions of observability, alarms, diagnosis and diagnosability.
- Chapter 8 presents the COMPASS workflow based on contract-based development.
- Finally, Chapter 9 contains a list of recommendations and advice that will help the reader address the most common pitfalls that can be encountered.

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

The COMPASS Approach

This chapter describes the use of the COMPASS toolset [1] in the development process. We remark that, in general, there may exist several workflows that are compatible with the use of COMPASS. Here, we illustrate the basic ingredients of a typical workflow as an example, and motivate it with reference to ESA standards. Then, in the rest of this tutorial we will show how COMPASS can be used operationally, to implement this workflow.

3.1 Space System Engineering

ESA and related institutions develop and maintain a series of standards for the management, engineering and product assurance in space projects and applications, known as ECSS. Among others, Standard ECSS-E-ST-10C [4] specifies the system engineering implementation requirements for space systems and space products development. More concretely, it states that

> "Systems engineering is defined as an interdisciplinary approach governing the total technical effort to transform a requirement into a system solution. A system is defined as an integrated set of elements to accomplish a defined objective. These elements include hardware, software, firmware, human resources, information, techniques, facilities services, and other support elements."

Moreover [4] partitions system engineering into the following functions:

requirements engineering, which consists of requirements analysis and validation, requirements allocation, and requirements maintenance;

analysis, which is performed for the purpose of resolving requirements conflicts, decomposing and allocating requirements during functional analysis, assessing system effectiveness (including analyzing risk factors); and complementing testing evaluation and providing trade studies for assessing effectiveness, risk, cost and planning;

design and configuration, which results in a physical architecture, and its complete system functional, physical and software characteristics;

verification, whose objective is to demonstrate that the deliverables conform to the specified requirements, including qualification and acceptance;

system engineering integration and control, which ensures the integration of the various engineering disciplines and participants throughout all the project phases.

3.2 The COMPASS Approach

The COMPASS toolset addresses, in a coherent manner, different aspects that are relevant to the engineering of complex systems, such as co-engineering of hardware and software, performability and dependability, reliability, availability, maintainability and safety engineering (RAMS). COMPASS offers a multi-disciplinary approach that supports the early design phases by developing systems at an architecture level. Thus it mainly targets the "requirement engineering" and "analysis" functions of system engineering, but also tackles the "design and configuration" and "verification" phases.

More concretely, COMPASS provides a specification language that offers convenient ways to describe nominal hardware and software operation, hybridity, (probabilistic) faults and their propagation, error recovery, and degraded modes of operation. A system specification is hierarchically organized into components which interact through connections via ports allowing for both message (event) and continuous (data) communication, and which can be reconfigured dynamically. The specification formalism is inspired by the Architecture Analysis and Design Language AADL [10] and its Error Model Annex [11]. It is named *System-Level Integrated Modeling (SLIM) Language.*

SLIM is equipped with a formal semantics that opens up the possibility to apply a wealth of formal methods for various kinds of verification and validation activities. The latter are supported by an integrated toolset that supports the following functionalities (see Figure 3.1 for an overview). The list also provides references to the corresponding ECSS standards.

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

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

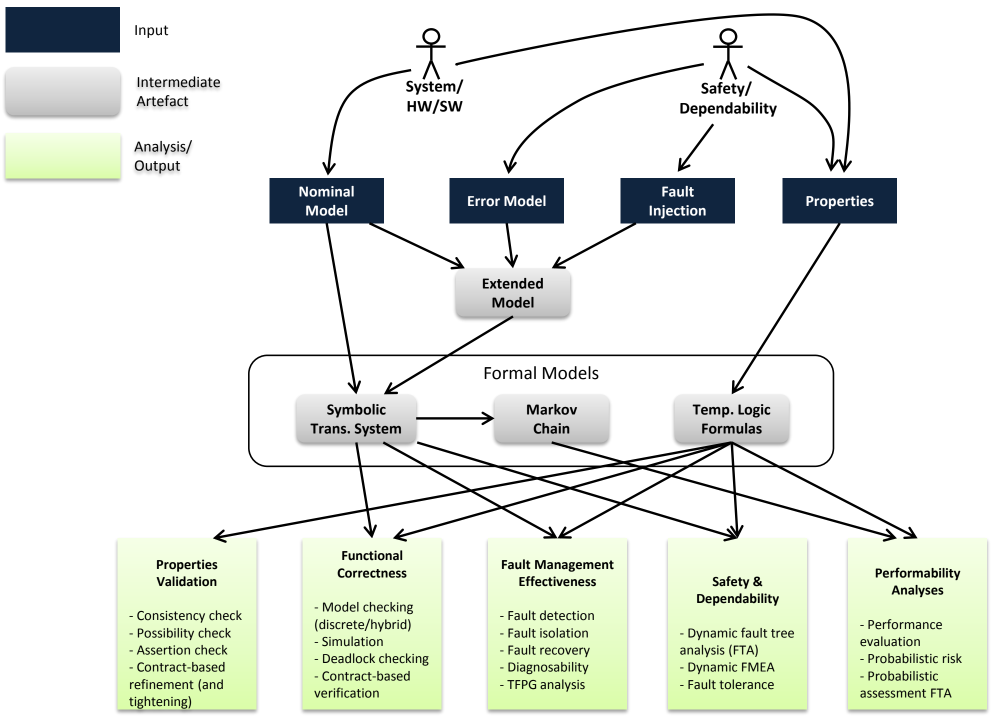


Figure 3.1: Overview of COMPASS Toolset Functionalities

Safety and Dependability Analysis [7, 5, 9, 8]: Analyzing system safety and dependability is a fundamental step that is performed in parallel with system design and verification of functional correctness. The goal is to investigate the behavior of a system in degraded conditions (that is, when some parts of the system are not working properly, due to malfunctions) and to ensure that the system meets the safety requirements that are required for its deployment and use. Key techniques in this area are (dynamic) fault tree analysis, (dynamic) Failure Modes and Effects Analysis (FMEA), fault tolerance evaluation, and criticality analysis.

Performability Analysis [6]: To guarantee the required system performance in the presence of faults, integrated hardware and software models can be evaluated with respect to their performance behavior in degraded modes of operation. In line with the approach for the functional correctness, again model checking techniques are employed for assessing this type of requirements.

Fault Detection, Identification and Recovery Analysis [5]: System models can include a formal description of both the fault detection and identification sub-systems, and the recovery actions to be taken. Based on these models, tool facilities are provided to analyze the operational effectiveness of the (probabilistic) FDIR measures, and to investigate the observability requirements that make the system diagnosable.

In summary, the overall process of analyzing system specifications involves the following steps:

1. SLIM specifications (describing the nominal and, if applicable, the error behavior) are entered using a text editor, and are saved to one or more files.
2. Files are loaded into the toolset.
3. If error behavior has to be considered, nominal and error models are related by means of fault injections.
4. To interactively explore the dynamic behavior of the system, the model simulation feature of the toolset can be employed.
5. Some of the subsequent analyses require writing properties. COMPASS offers several ways to specify such properties.
6. Finally, depending on the type of the system, a plethora of analyses can be applied.

In the rest of this tutorial, we show how COMPASS can be used to implement these steps and we illustrate the analyses that it supports.

Chapter 4

COMPASS as a Model Checker

In this chapter, we explain the basics of the COMPASS toolset: how to run COMPASS, how to write and manage models in SLIM, how to specify properties, and how to run simulations and to verify properties.

4.1 Writing the First Model

We use the *battery sensor* model, for exemplification. The model architecture is illustrated in Figure 4.1. It consists of a redundant pair of sensors powered by a redundant pair of

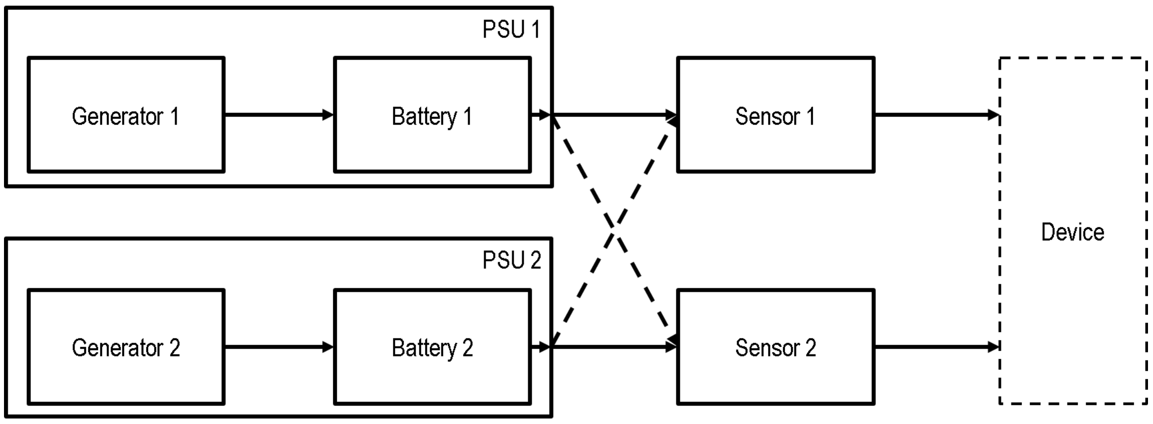


Figure 4.1: Battery sensor model.

batteries, charged by dedicated generators. A pair generator/battery is called a PSU (Power System Unit). A hypothetical device that depends on these sensor readings is assumed to be operational (alive) provided that at least one sensor is working correctly (providing a reading). A generator failure causes a permanent loss of power supply, and a sensor failure causes a permanent loss of its reading. In absence of power supply, a battery starts discharging; when depleted, the corresponding sensor stops working. The battery sensor model has three modes: *primary*, $secondary_1$ and $secondary_2$. In primary mode, sensors are powered by the corresponding battery they are initially connected to; in case of failure of one battery, the system may be reconfigured (secondary modes 1 and 2 – connections indicated with dotted lines), so that both sensors are powered by the same battery (the one that did not fail).

COMPASS models are written in the SLIM language. For a complete account of SLIM, we refer the reader to the user manual [3], and the syntax and semantics document [2], both included in the COMPASS distribution.

The models presented here can be found in the `documentation/examples/battery_sensor` folder – see models `system_discrete_simple.slim`, `system_discrete.slim` and `system.slim` and the other accompanying files. We remark that, below, we introduce these models in successive steps, therefore some parts may be omitted and described later on in this tutorial.

SLIM enables the separate definition of the *interface* and *implementation* of component models. We start by defining the interface of some basic components. Figure 4.2 defines the interface of the *generator*, *battery* and *sensor* components. We use the keyword **system** that models a generic component, note however that SLIM enables the definition of ad-hoc software or hardware component types.

The generator is simply a source of power; this is modeled by means of a (Boolean) output data port, named `has_power`, which is true by default (we remark that, for the time being, we are modeling the *nominal* behavior; faults will be introduced later, in Chapter 5). The battery and sensor are modeled in a similar fashion. They both have an input data port for modeling input power. The battery also has an input data port called `zero_consumption` (modeling no consumption when the battery is disconnected from the sensors) and a `low` output data port (true if the charge level of the battery is low). The sensor has an output data port `reading` that is true if the sensor is providing a reading.

```
system Generator
  features
    has_power: out data port bool {Default => "true";};
end Generator;

system Battery
  features
    has_power_out: out data port bool {Default => "true";};
    has_power_in: in data port bool {Default => "true";};
    zero_consumption: in data port bool {Default => "false";};
    low: out data port bool {Default => "false";};
end Battery;

system Sensor
  features
    reading: out data port bool {Default => "true";};
    has_power: in data port bool {Default => "true";};
end Sensor;
```

Figure 4.2: Battery sensor model: basic components.

Figure 4.3 models the implementation of the components described so far. The generator implementation does not contain further specifications, whereas the battery and sensor implementations specify the behavior of the corresponding components using states and transitions. A battery has three possible states (modeling the charge level): `full`, `low_` and `empty`. The initial state is `full`. A transition models a state change, which may be triggered by an input event, may be conditioned by an input condition and may also define an output effect. For instance, the second transition models the fact the a fully charged battery, in absence of input power and when connected to a sensor, can move to state `low_`; as an effect, the battery still has output power, but the `low` output port becomes true. Similarly for the other transitions. The sensor implementation defines a simple state machine with only one state and two possible transitions; depending on whether input power is present or not, the `reading` output is set to true or false.

```
system implementation Generator.Imp
end Generator.Imp;

system implementation Battery.Imp
  states
    full: initial state;
    low_: state;
    empty: state;
  transitions
    full -[ when has_power_in or zero_consumption
            then has_power_out := true ]-> full;
    full -[ when not has_power_in and not zero_consumption
            then has_power_out := true; low := true ]-> low_;
    low_ -[ when zero_consumption
            then has_power_out := true ]-> low_;
    low_ -[ when not zero_consumption
            then has_power_out := false ]-> empty;
    empty -[ then has_power_out := false ]-> empty;
end Battery.Imp;

system implementation Sensor.Imp
  states
    base: initial state;
  transitions
    base -[ when has_power then reading := true ]-> base;
    base -[ when not has_power then reading := false ]-> base;
end Sensor.Imp;
```

Figure 4.3: Battery Sensor model: basic components' implementation.

Figure 4.4 models the PSU component. A PSU (Power System Unit) groups a generator and a battery. The interface of a PSU specifies an input data port `zero_consumption`, with the same meaning as for the battery, and an output data port `has_power`, modeling the presence of output power.

The generator and the battery are connected together in the component implementation. A PSU implementation instantiates two sub-components of type `Battery.Imp` and `Generator.Imp`. The two sub-components are connected in such a way that the output power of the generator is connected with the input power of the battery, and the output power of the battery is connected with the output power of the PSU. Port `zero_consumption` is connected with the corresponding port of the battery.

```
system PSU
  features
    has_power: out data port bool;
    zero_consumption: in data port bool {Default => "false";};
end PSU;

system implementation PSU.Imp
  subcomponents
    battery: system Battery.Imp;
    generator: system Generator.Imp;
  connections
    port generator.has_power -> battery.has_power_in;
    port battery.has_power_out -> has_power;
    port zero_consumption -> battery.zero_consumption;
end PSU.Imp;
```

Figure 4.4: Battery sensor model: PSU.

Figure 4.5 puts all components together into a complete system. The system interface has the following data ports: `is_alive` tracks whether the system is alive (at least one sensor is providing an output), whereas `mode_selector` is an enumerative output data port that models the system configuration (primary or secondary). Moreover, there are two input event ports called `go_to_secondary1` and `go_to_secondary2` that models input events that trigger system re-configuration.

The system implementation instantiates two instances of the sensor component and two instances of the PSU component. Furthermore, it defines three different *modes*, namely `primary`, `secondary1` and `secondary2`. Modes are in spirit similar to states for sub-components, but it has to be remarked that states cannot be used within composite components, but only in leaf components. The connections link the output power of the two PSU with the input power of the two sensors. The transition relation specifies mode re-configuration, which has also the effect of changing `mode_selector`. It is interesting to remark that the connections are mode dependent: for instance, in mode `secondary1`, component `psu1` feeds both sensors. Moreover, the **`flow`** section defines the value of `is_alive` (true if at least one sensor provides a reading) and `zero_consumption` of the two PSU (true, respectively, in mode `secondary2` and `secondary1`).

Finally, Figure 4.6 encloses the overall system into a further component called `Enclosure` which exports only the re-configuration events at the interface. Note that these events are left free, hence they can be triggered by the external environment[1].

[1]We will come back to this point in Chapter 7, when we connect the system with a controller that can re-configure the system.

```
system System
  features
    is_alive: out data port bool;
    mode_selector:
      out data port enum(Primary, Secondary1, Secondary2)
      {Default => "Primary";};
    go_to_secondary1: in event port;
    go_to_secondary2: in event port;
end System;

system implementation System.Imp
  subcomponents
    sensor1: system Sensor.Imp;
    sensor2: system Sensor.Imp;
    psu1: system PSU.Imp;
    psu2: system PSU.Imp;
  connections
    port psu1.has_power -> sensor1.has_power
      in modes (primary, secondary1);
    port psu1.has_power -> sensor2.has_power
      in modes (secondary1);
    port psu2.has_power -> sensor1.has_power
      in modes (secondary2);
    port psu2.has_power -> sensor2.has_power
      in modes (primary, secondary2);
    flow sensor1.reading or sensor2.reading -> is_alive;
    flow true -> psu1.zero_consumption in modes (secondary2);
    flow true -> psu2.zero_consumption in modes (secondary1);
  modes
    primary: initial mode;
    secondary1: mode;
    secondary2: mode;
  transitions
    primary -[ go_to_secondary1
      then mode_selector := Secondary1 ]-> secondary1;
    primary -[ go_to_secondary2
      then mode_selector := Secondary2 ]-> secondary2;
end System.Imp;
```

Figure 4.5: Battery sensor model: system level.

4.1.1 Adding More States

We modify the previous model, introducing additional states and transitions, since we want to specify a battery may discharge at different rates, depending on whether it is powering one or two sensors. For simplicity, we then split the `low_` state of a battery into `low_single` and

```
system Enclosure
  features
    go_to_secondary1: in event port;
    go_to_secondary2: in event port;
end Enclosure;

system implementation Enclosure.Imp
  subcomponents
    sys: system System.Imp;
  connections
    port go_to_secondary1 -> sys.go_to_secondary1;
    port go_to_secondary2 -> sys.go_to_secondary2;
end Enclosure.Imp;
```

Figure 4.6: Battery sensor model: top-level enclosure.

`low_double`. Similarly, we introduce a port called `double_consumption`, that drives the state of the battery. The battery and PSU interfaces are extended with this port (see Figure 4.7).

```
system Battery
  features
    [...]
    double_consumption: in data port bool {Default => "false";};
end Battery;

system PSU
  features
    [...]
    double_consumption: in data port bool {Default => "false";};
end PSU;
```

Figure 4.7: Battery sensor model: double consumption ports.

The new battery states `low_single` and `low_double` are declared, and the transition relation is extended accordingly (see Figure 4.8): the value of the `double_consumption` input port controls whether the battery mode changes from `full` to `low_single` or `low_double`. The PSU implementation requires drawing one more connection (see Figure 4.9). Finally, the system implementation (Figure 4.10) sets the value of the `double_consumption` port depending on the system mode.

4.2 Adding Delays

We extend the battery sensor model, using clocks to model temporal delays for the discharging of the batteries and for failure propagation in the sensors. This extension enables a more faithful modeling of the dynamics of the underlying system.

```
system implementation Battery.Imp
  states
    full: initial state;
    low_single: state;
    low_double: state;
    empty: state;
  transitions
    full -[ when has_power_in or zero_consumption
            then has_power_out := true ]-> full;
    full -[ when not has_power_in and
            not double_consumption and
                 not zero_consumption
            then has_power_out := true; low := true ]-> low_single;
    full -[ when not has_power_in and
            double_consumption and
                 not zero_consumption
            then has_power_out := true; low := true ]-> low_double;
    low_single -[ when zero_consumption
                  then has_power_out := true ]-> low_single;
    low_single -[ when not zero_consumption
                  then has_power_out := false ]-> empty;
    low_double -[ when zero_consumption
                  then has_power_out := true ]-> low_double;
    low_double -[ when not zero_consumption
                  then has_power_out := false ]-> empty;
    empty -[ then has_power_out := false ]-> empty;
end Battery.Imp;
```

Figure 4.8: Battery sensor model: double consumption.

```
system implementation PSU.Imp
  [...]
  connections
    [...]
    port double_consumption -> battery.double_consumption;
end PSU.Imp;
```

Figure 4.9: Battery sensor model: double consumption port connection.

The sensor model implementation (Figure 4.11) is extended by declaring a clock called `delay`. The state of the sensor is modified so that it is updated every time unit; this is realized by adding an *invariant* `delay <= 1` that has to be true in the `base` state and conditioning the transitions with the condition `delay >= 1`, that has to be true when triggering the transition. The combined effect of the invariant and the condition, in this model, is that the transitions will be taken precisely every time unit. Notice that, upon taking a transition, the assignment

```
system implementation System.Imp
  [...]
  connections
    [...]
    flow true -> psu1.double_consumption in modes (secondary1);
    flow true -> psu2.double_consumption in modes (secondary2);
  [...]
end System.Imp;
```

Figure 4.10: Battery sensor model: double consumption definition.

`delay := 0` restarts the clock.

```
system implementation Sensor.Imp
  subcomponents
    delay: data clock;
  states
    base: initial state while (delay <= 1);
  transitions
    base -[ when delay >= 1 and has_power
            then delay := 0; reading := true ]-> base;
    base -[ when delay >= 1 and not has_power
            then delay := 0; reading := false ]-> base;
end Sensor.Imp;
```

Figure 4.11: Battery sensor model: sensor delay.

We do a similar thing for the battery model implementation (Figure 4.12). Here, we use different delays for different states of the battery: in mode `full` the battery will take 3 time units before the charge level becomes low. Moreover, depending on the whether the battery is connected to one or two sensors, the battery will discharge with different rates (2 time units or 1 time unit, are needed, respectively, to reach the `empty` state).

4.3 Running the Toolset

The COMPASS toolset provides a graphical user interface (GUI) written in python. The executable of the GUI can be found in the `scripts` directory. For a quick start, the GUI can be run as follows:

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

This launches the main window of the COMPASS toolset (see Figure 4.13).

It requires the installation of Python version 2.7 or higher – plus other libraries. For an exhaustive list of all the requirements we refer to Section 3.1 in the user manual.

The GUI executable accepts also many command-line options – see Section 3.5 in the manual for a complete list.

```
system implementation Battery.Imp
  subcomponents
    delay: data clock;
  states
    full: initial state while (delay <= 3);
    low_single: state while (delay <= 2);
    low_double: state while (delay <= 1);
    empty: state while (delay <= 1);
  transitions
    full -[ when delay >= 1 and (has_power_in or zero_consumption)
           then delay := 0 ; has_power_out := true ]-> full;
    full -[ when delay >= 3 and
                 not has_power_in and
                 not double_consumption and
                 not zero_consumption
           then delay := 0;
                 has_power_out := true;
                 low := true ]-> low_single;
    full -[ when delay >= 3 and
                 not has_power_in and
                 double_consumption and
                 not zero_consumption
           then delay := 0;
                 has_power_out := true;
                 low := true ]-> low_double;
    low_single -[ when delay >= 1 and zero_consumption
                  then delay := 0;
                  has_power_out := true ]-> low_single;
    low_single -[ when delay >= 2 and not zero_consumption
                  then delay := 0; has_power_out := false ]-> empty;
    low_double -[ when delay >= 1 and zero_consumption
                  then delay := 0; has_power_out := true ]-> low_double;
    low_double -[ when delay >= 1 and not zero_consumption
                  then delay := 0; has_power_out := false ]-> empty;
    empty -[ when delay >= 1
             then delay := 0; has_power_out := false ]-> empty;
end Battery.Imp;
```

Figure 4.12: Battery sensor model: battery delay.

4.4 Loading and Saving Models

The *Model* pane of the toolset is displayed on startup (as depicted in Figure 4.13).

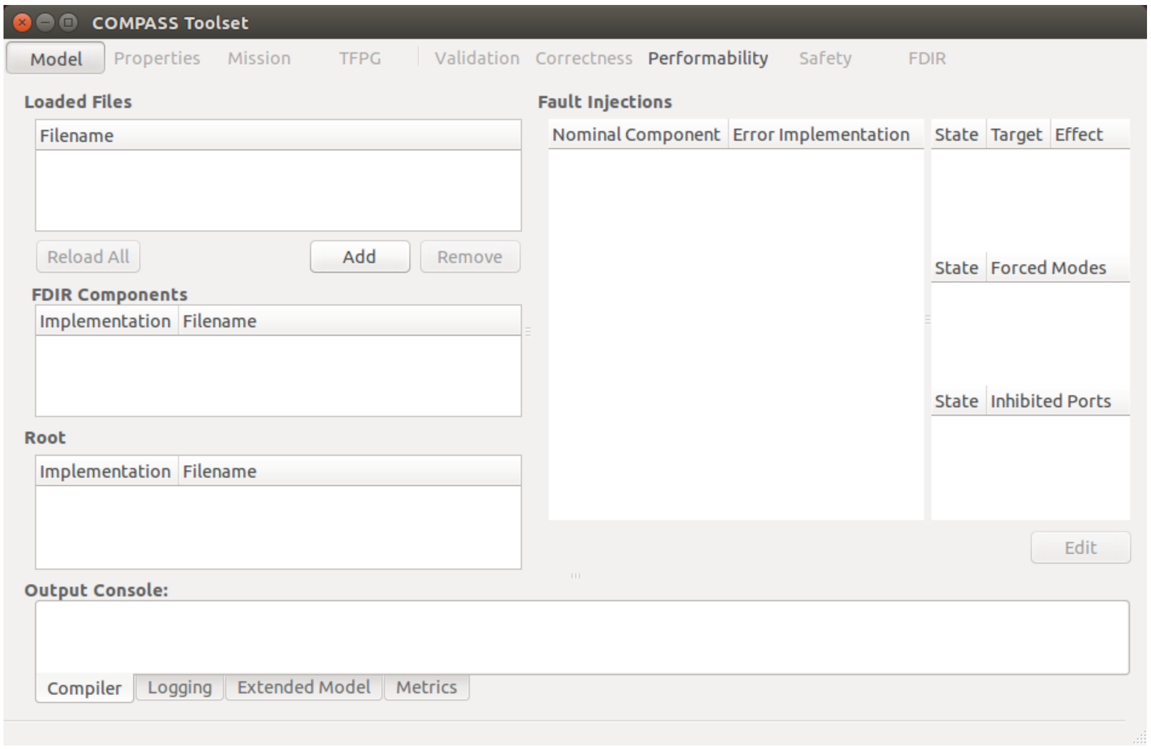


Figure 4.13: Main window of the COMPASS Toolset.

4.4.1 Loading Models

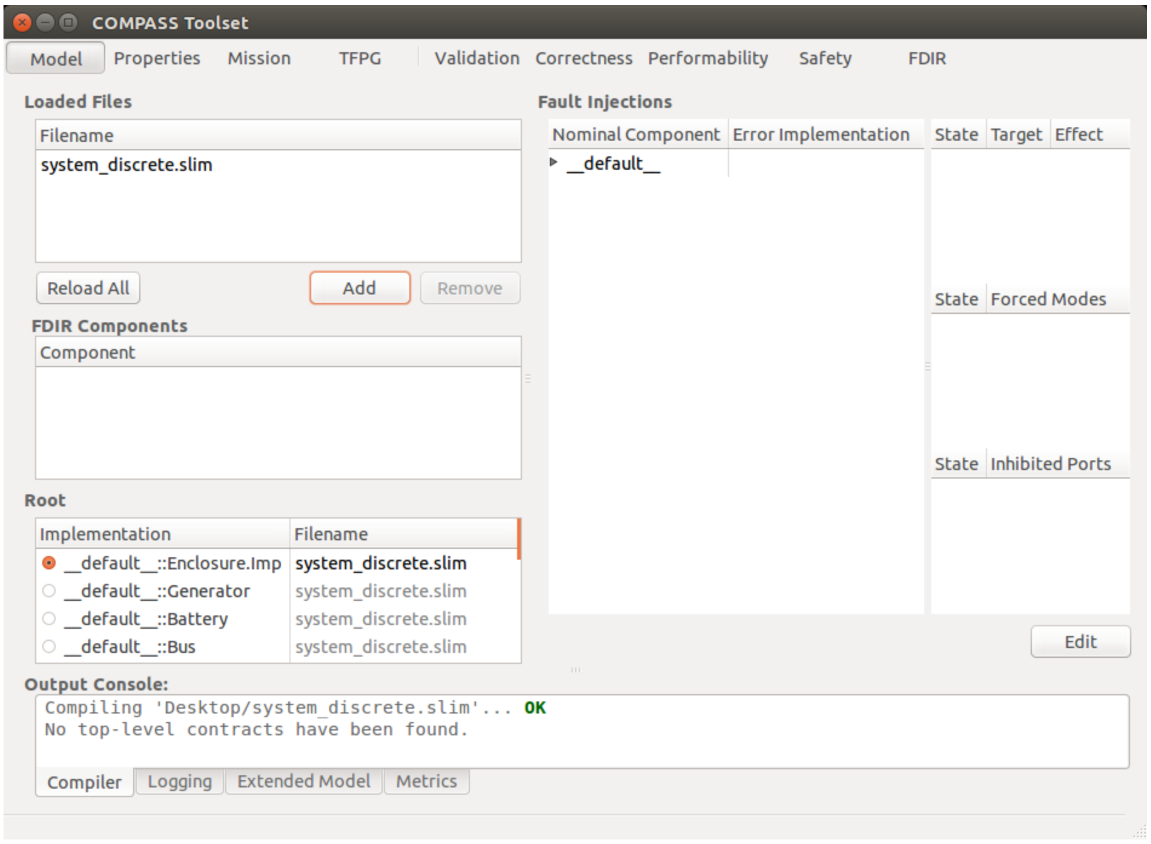
A SLIM model can be edited using an external editor. Once edited, it can be loaded using the *Add* button. Let us now load the `system_discrete` model in the GUI of the COMPASS toolset, by going through the following steps.

Steps

- Load the GUI
- Click the *Add* button below the *Loaded Files* section in the window
- The *Open SLIM Files* window opens
- Select the file `documentation/examples/battery_sensor/system_discrete.slim`
- Click *Open*
- On the *Output Console* a message confirming the loading is shown. The different root components appear in the *Root* pane; the root `__default__::Enclosure.Imp` is selected by default (see Figure 4.14) (`__default__` is used as a prefix for components not inside a package).

4.4.2 Saving Models

A model can be saved as follows.



Filename
system_discrete.slim

Nominal Component	Error Implementation
__default__	

Implementation	Filename
[✓ __default__::Enclosure.Imp]	system_discrete.slim
__default__::Generator	system_discrete.slim
__default__::Battery	system_discrete.slim
__default__::Bus	system_discrete.slim

```
Compiling 'Desktop/system_discrete.slim'... OK
No top-level contracts have been found.
```

Figure 4.14: Main Window of the COMPASS toolset with model `system_discrete` loaded.

Steps

- Load the GUI and a model
- Open the *File* menu
- Select the *Save SLIM Model as...* button; a dialog pops up asking for the name of the file to save the model to (see Figure 4.15)
- Click *Save*; the specified file has been created

4.5 Simulating a Model

Simulating a model is typically a useful starting point to check that its behavior corresponds to the intended one. The COMPASS toolset allows the user to carry out three different simulation activities:

- *random* simulation, in which the values of the model signals are chosen automatically
- *guided by transition* simulation, in which the user can explicitly force the system's behavior by picking one of the available transitions

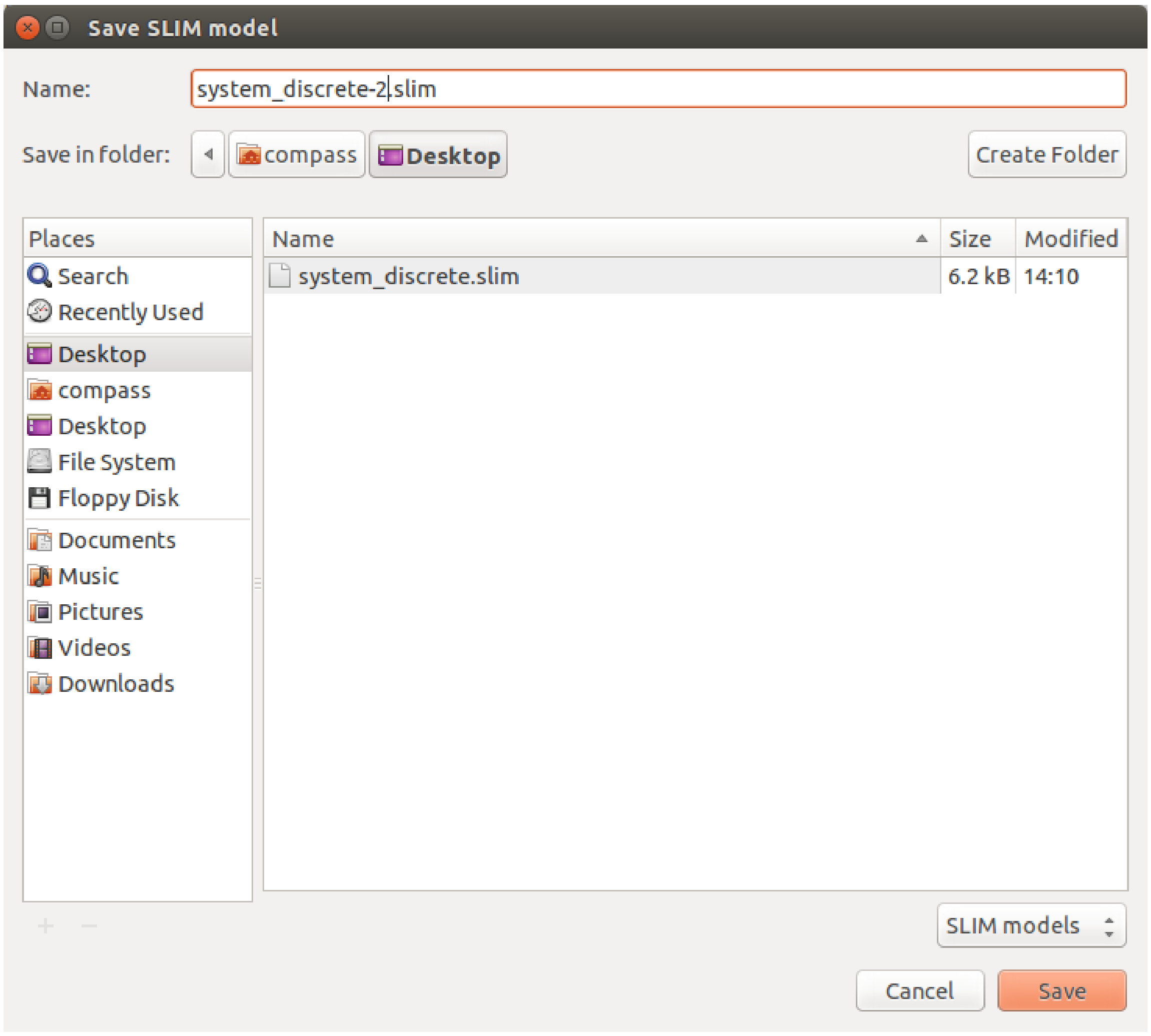


Figure 4.15: Pop-up window to save a model.

- *guided by values* simulation, in which the values of the model signals can be forced by the user in order to verify the behavior of the system under specific scenarios

Let us run a random simulation of the `system_discrete` example, as follows.

Steps

- Load the model as shown in Section 4.4.1
- Click on the *Correctness* tab; the *Model Simulation* pane is shown
- Select *Random* as type of simulation and 10 as *Length*
- Click the *Run* button
- A trace like the one shown in Figure 4.16 is displayed as a table where the first column provides the name of the model elements and the remaining columns give their values, for each step of the simulation. Boolean stripes are shown with a waveform with two values, the "high" one for true and the "low" one for the false.

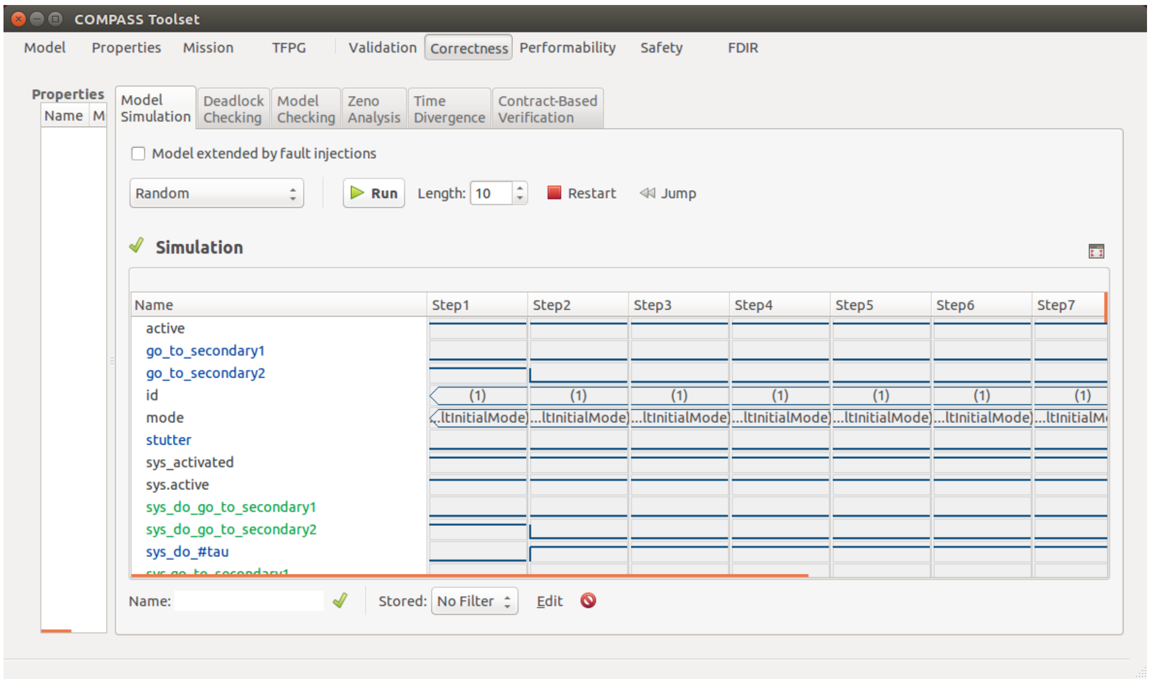


Figure 4.16: Random simulation trace created for the `system_discrete` example.

We refer to Section 9.4.1 and 9.4.2 of the user manual for a more detailed explanation about traces and the other types of simulation.

4.6 Writing a Property

The COMPASS interface provides different ways to write properties. They provide different levels of assistance, ranging from requiring only some design parameters to be specified (CSSP) to inputting formal properties directly. At the basic level, formal properties can be specified directly using the SLIM expression language. This provides the most flexible approach, but requires a strong technical knowledge. Property patterns provide a fixed structure, depending on a few parameters, in which placeholders can be filled in to specify some formal property, accompanied by a human readable description. Finally, the CSSP (Catalogue of System and Software Properties) can be used, which simply requires some model parameters to be filled in, such as time delays or expected ranges of port values.

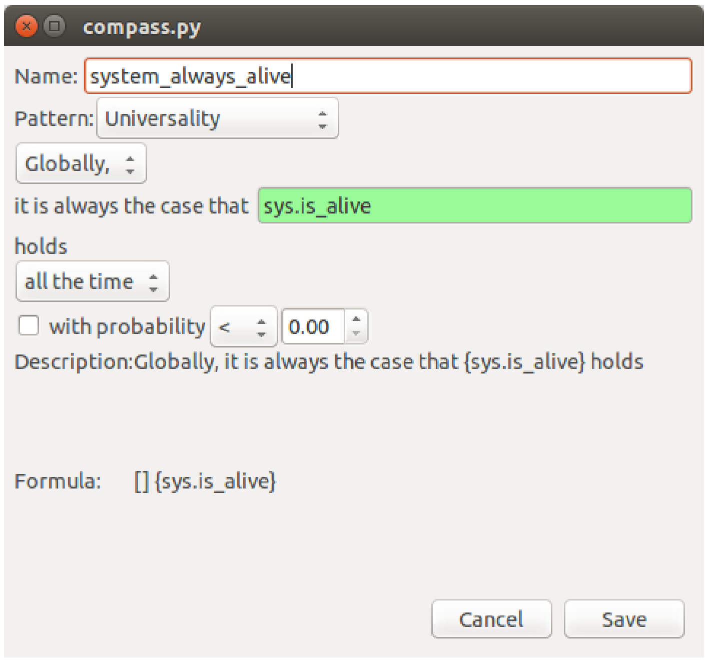
All three kinds properties can be specified from the *Properties* pane under the *Properties* tab. Furthermore, the *Requirements* pane provides a *Wizard* button that helps with a step-by-step approach to specifying properties.

For the purpose of this tutorial, let us now define a property for the `system_discrete` example using property patterns. We want to specify that the system is always alive, that is, the atomic proposition `sys.is_alive` always holds. We do so as follows.

Steps

- Load the model as shown in Section 4.4.1
- Click on the *Properties* tab; the *Requirements* pane is shown.

- On the left, select the component `Enclosure.Imp`
- Click on the *Properties* sub-pane
- Click the bottom-most *Add* button; define the property `system_always_alive` as shown in figure 4.17
- Click the button *Save*; the name of the properties appears in the bottom-most pane, and a "(1)" is shown next to the name `Enclosure.Imp` on the left to specify that one property has been added for that component (see Figure 4.18)



Name: system_always_alive
Pattern: Universality
Globally,
it is always the case that sys.is_alive
holds
all the time
with probability < 0.00
Description:Globally, it is always the case that {sys.is_alive} holds

Formula: [] {sys.is_alive}

Figure 4.17: Adding the property `system_always_alive` for the `system_discrete` example.

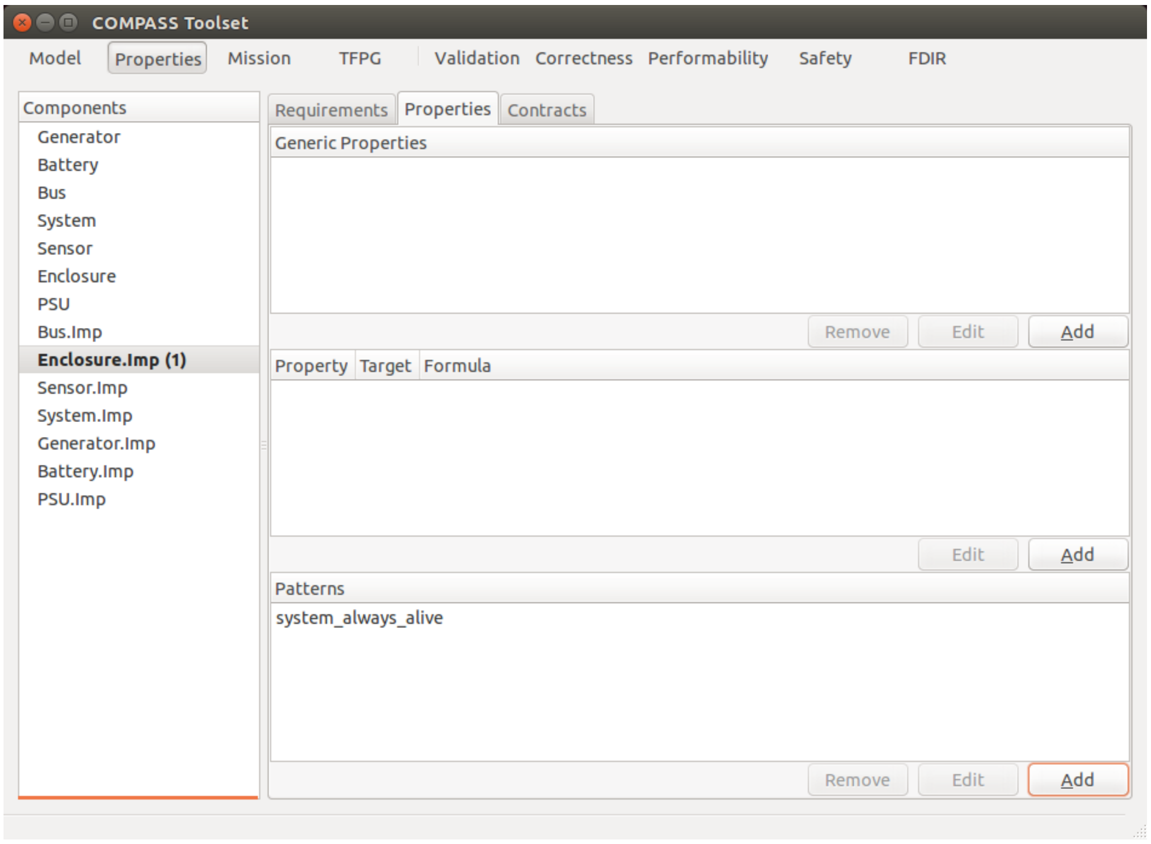
4.7 Saving Properties

Properties are saved along with the model, in the same file(s). For instance, let us save the `system_discrete` model, with the property we have just defined, as follows.

Steps

- Click the *File* menu
- Select item *Save SLIM model as ...*
- Insert the name `system_discrete_with_prop.slim` and save the modified model to the desktop

If we open the model we have saved, we can see that it includes the property, see Figure 4.19.



Generator
Battery
Bus
System
Sensor
Enclosure
PSU
Bus.Imp
Enclosure.Imp (1)
Sensor.Imp
System.Imp
Generator.Imp
Battery.Imp
PSU.Imp

system_always_alive

Figure 4.18: *Properties* pane after the property `system_always_alive` has been added.

```
...
system implementation Enclosure.Imp
subcomponents
    sys: system System.Imp;
connections
    port go_to_secondary1 -> sys.go_to_secondary1;
    port go_to_secondary2 -> sys.go_to_secondary2;
properties
    Patterns => ([ Name => "system_always_alive"; Pattern =>
      "Globally, it is always the case that {sys.is_alive} holds "; ]);
end Enclosure.Imp;
...
```

Figure 4.19: Battery sensor model with one property.

4.8 Loading Properties

Properties are loaded along with the model, since they are stored in the same file(s). For backward compatibility, it is possible to load properties from an xml file. For instance, model `system_discrete` comes with a property file `system_discrete.propxml`, that can be loaded as follows.

Steps

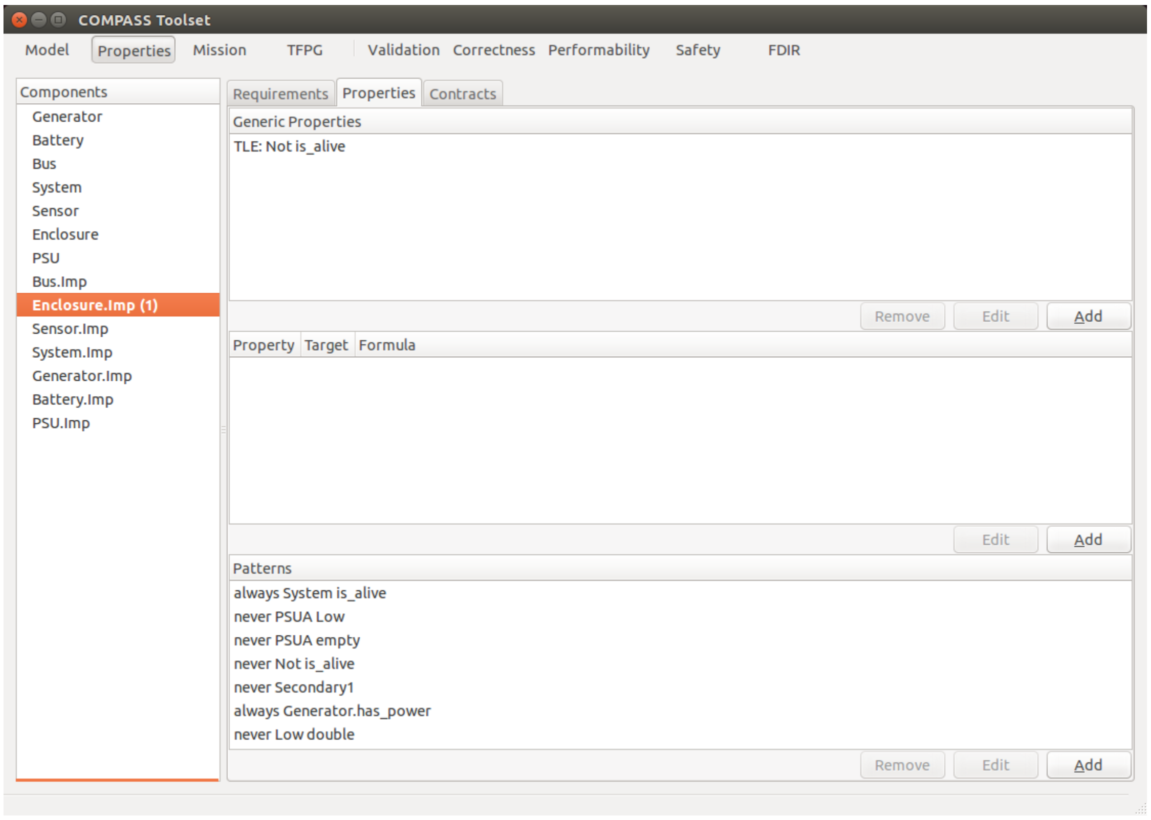
- Load the model as shown in Section 4.4.1
- Click on the *Properties* tab; the *Requirements* pane is shown (empty)
- Click on the *File* menu
- Select item *Load Properties ...* and load properties in file `system_discrete.propxml`
- Click on the *Properties* sub-pane
- On the left, the *#Prop.* column shows that 16 properties have been loaded for component `Enclosure.Imp`
- Click on the item `Enclosure.Imp` (see Figure 4.20)
- The following generic properties have been loaded:
  - `TLE: Not is_alive`

  Under the *Pattern properties* sub-pane, the following properties are present:
  - `always System is_alive`
  - `never PSUA Low`
  - `never PSUA empty`
  - ...

4.9 Model Checking a Property

Model checking is used to check a SLIM model against one or more properties. The purpose of model checking is to formally verify that all the behaviors admitted by the model do verify the properties at hand and, if not, generate an execution trace that shows a violation (i.e, counterexample). A counterexample is useful to identify the reason for a violation and understand whether the it is an expected or an unexpected behavior of the system. If the violation is unexpected, it may be either be due to a flaw in the system specification, or a mistake in the encoding of the system as a formal model. In the former case, this may trigger a change in the system specification (and in the model, as a consequence), in the latter case it probably requires a revision of the formal model.

In order to run property verification, it is first required to define some properties in the *Properties* tab, or to load a model already containing some properties. Depending on the semantics of the model (finite or infinite), different engines can be selected and are properly enabled or disabled. For each engine, different options may be selected; see Section 9.4.4 in the user manual for a complete description of all the choices. In the following, we show an example using the `system_discrete` model.



TLE: Not is_alive

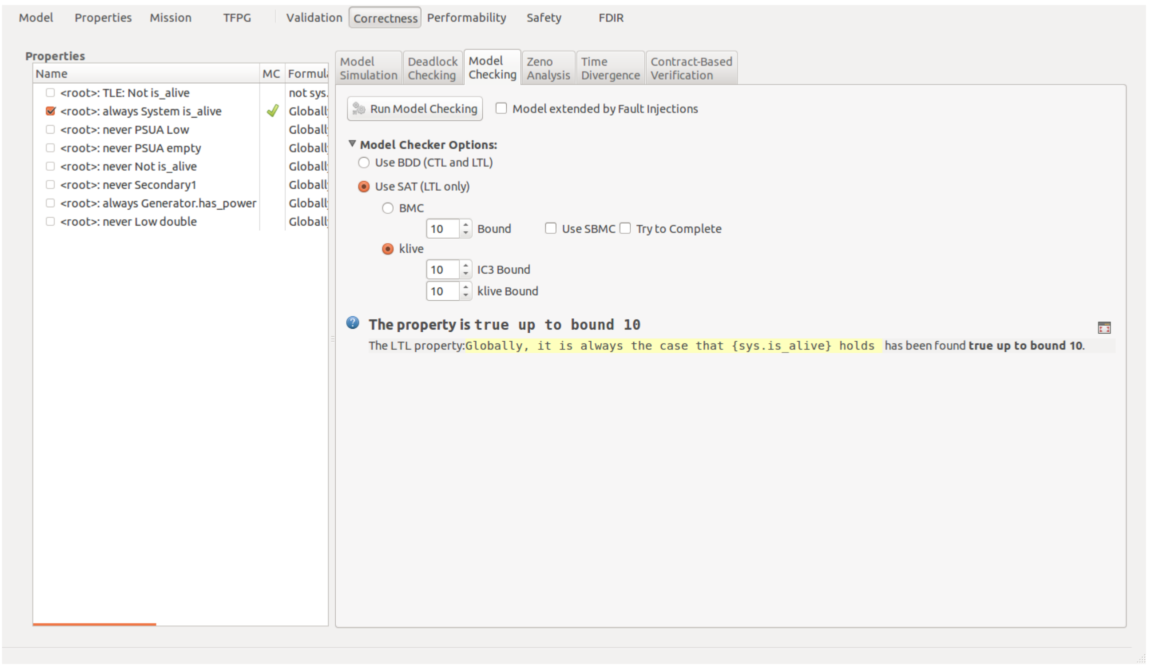
always System is_alive
never PSUA Low
never PSUA empty
never Not is_alive
never Secondary1
always Generator.has_power
never Low double

Figure 4.20: List of properties in `system_discrete.propxml` file.

Steps

- Load the model as shown in Section 4.4.1
- Open the *File* menu and load the properties contained in file `documentation/examples/battery_sensor/system_discrete.propxml`
- Click on the *Correctness* tab and select the *Model Checking* pane
- The following properties are shown on the left *Properties* pane:
  - `TLE: Not is_alive`
  - `always System is_alive`
  - `never PSUA Low`
  - `never PSUA empty`
  - `never Not is_alive`
  - `never Secondary1`
  - `always Generator.has_power`
  - `never Low double`
- Select the property `always System is_alive`; this property states that the variable `is_alive` is always true.

- Disable the *Model Extended by Fault Injections* checkbox
- Expand *Model Checker Options* and select the *klive* engine
- Click the *Run Model Checking* button
- The property results to be true (up to bound), as shown in Figure 4.21. The reason is that we are considering only the nominal behavior of the model; in Section 5.7 we run the same analysis considering also the error model.



Properties

Name	MC	Formul
☐ <root>: TLE: Not is_alive		not sys.
☑ <root>: always System is_alive	✓	Globall
☐ <root>: never PSUA Low		Globall
☐ <root>: never PSUA empty		Globall
☐ <root>: never Not is_alive		Globall
☐ <root>: never Secondary1		Globall
☐ <root>: always Generator.has_power		Globall
☐ <root>: never Low double		Globall

☐ Model extended by Fault Injections

Model Checker Options:
○ Use BDD (CTL and LTL)
● Use SAT (LTL only)
○ BMC
10 Bound ☐ Use SBMC ☐ Try to Complete
● klive
10 IC3 Bound
10 klive Bound

The property is `true up to bound 10`
The LTL property:`Globally, it is always the case that {sys.is_alive} holds` has been found **true up to bound 10.**

Figure 4.21: Confirmation of the property `always System is_alive` of the `system_discrete` example.

4.10 Deadlock Checking

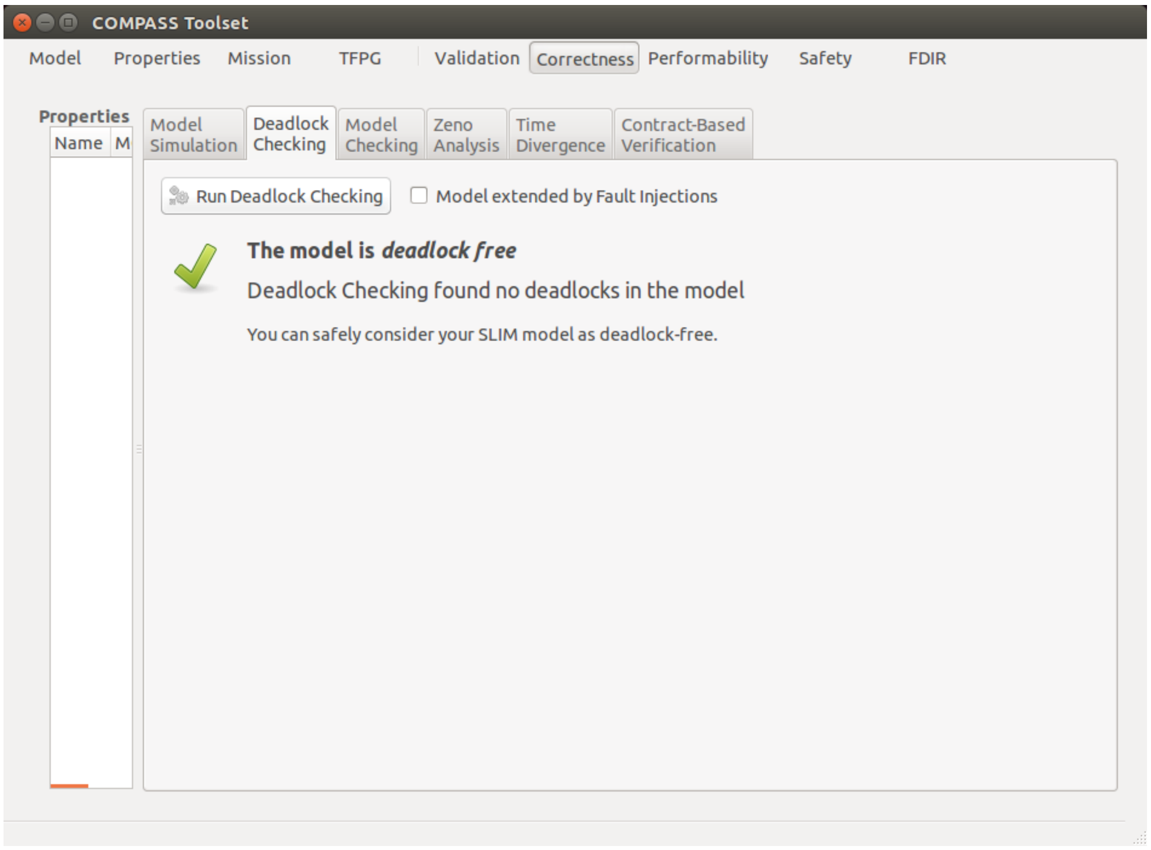
Deadlock checking is a useful analysis that allows the user to check for the presence of deadlocks in a specific SLIM model. The two possible outcomes (presence of deadlocks or absence of deadlocks) are shown to the user by different messages that appear on the pane; Figure 4.22 shows that no deadlock are present in the `system_discrete` model. The steps to reproduce this result are as follows.

Deadlock checking is important as it ensures the correctness of the outcome of model checking. If there exists a deadlock in the model, some problems may not be found during model checking. This is described in more detail in Section 9.1 and in the user manual.

Steps

- Load the model as shown in Section 4.4.1

- Click on the *Correctness* tab
- Select the *Deadlock Checking* pane
- Click the *Run* button



The model is *deadlock free*

Deadlock Checking found no deadlocks in the model

You can safely consider your SLIM model as deadlock-free.

Figure 4.22: Deadlock checking for the `system_discrete` example.

Chapter 5

Dealing with Faults

In this chapter, we explain how to deal with faults: specifying faults and fault events, and how to automatically extend a model with fault definitions. We also introduce Timed Failure Propagation Graphs (TFPGs), a formalism to model failure propagation.

5.1 Writing an Error Model

The first step is to define one or more error models. An error model defines possible errors that may affect the nominal behavior of the system we are modeling. As for nominal models, the specification of an error model comprises an interface and an implementation. We define a simple error model for the battery sensor example, modeling a permanent failure, in Figure 5.1.

The interface of the error model defines a set of error states, `ok` and `dead` in this example; `ok` is an activation state, that is, it is the default state upon initialization (and re-activation) of the corresponding component. The implementation of the error model defines *error events*, and transitions between error states, that may be guarded by error events. In this example we define one error event named `fault`, guarding a transition from state `ok` to state `dead`. The dynamics being permanent is evident from the fact that there is no outgoing transition from state `dead` – the fault cannot disappear or be repaired, unless the model is reactivated.

```
error model PermanentFailure
  features
    ok: activation state;
    dead: error state;
end PermanentFailure;

error model implementation PermanentFailure.Imp
  events
    fault: error event;
  transitions
    ok -[fault]-> dead;
end PermanentFailure.Imp;
```

Figure 5.1: Battery sensor model: permanent error model.

5.2 Adding Probabilities in Error Models

Often, the occurrence of faults can be estimated by statistics, usually by assuming that they occur with a certain probability. SLIM supports probabilities by means of attaching error rates in error models, which represent exponential distributions. An error rate associated with an error event means that the event may occur at a random time as determined by the exponential distribution.

We start with the discrete battery sensor model, and add an error rate to the error model as created in Section 5.1, as depicted in Figure 5.2. The error rate is set to 0.01 as per the keywords **`occurrence poisson`** `0.01`.

```
error model implementation PermanentFailure.Imp
  events
    fault: error event occurrence poisson 0.01;
  transitions
    ok -[fault]-> dead;
end PermanentFailure.Imp;
```

Figure 5.2: Battery sensor model: permanent error model with rate.

5.3 Modeling Fault Injections

A fault injection specifies how an error model is paired with a nominal model, and the consequence (effects) of an error on the behavior specified in the nominal specification.

We want to associate a permanent fault to generators and sensors, in the battery sensor example. We extend the implementation of the generator and sensor (nominal) models, as shown in Figure 5.3. A fault injection consists of two parts: the association with the error model (`PermanentFailure.Imp`) and a specification of the effects of the errors. For the generator model, we specify that, when the component is in state `dead`, the `has_power` output is set to **`false`** (no output power), whereas for the sensor model, we specify that, when the component is `dead`, its output reading is set to **`false`** (no reading).

It is also possible to edit fault injections using the GUI interface. For instance, let us show how to edit the effect of the fault injection for the generator.

Steps

- Load the model as shown in Section 4.4.1
- Click the *Edit* button
- In the pop-up window, expand the *_default_* element
- Select `Generator.Imp`
- Change the *Effect* value from **`false`** to **`true`**

```
system implementation Generator.Imp
  properties
        ErrorModel => classifier(PermanentFailure.Imp);
        FaultEffects => ([State=>"dead";
                          Target=>reference(has_power);
                          Effect=>"false";]);
end Generator.Imp;

system implementation Sensor.Imp
  [...]
  properties
        ErrorModel => classifier(PermanentFailure.Imp);
        FaultEffects => ([State=>"dead";
                          Target=>reference(reading);
                          Effect=>"false";]);
end Sensor.Imp;
```

Figure 5.3: Battery sensor model: permanent error model.

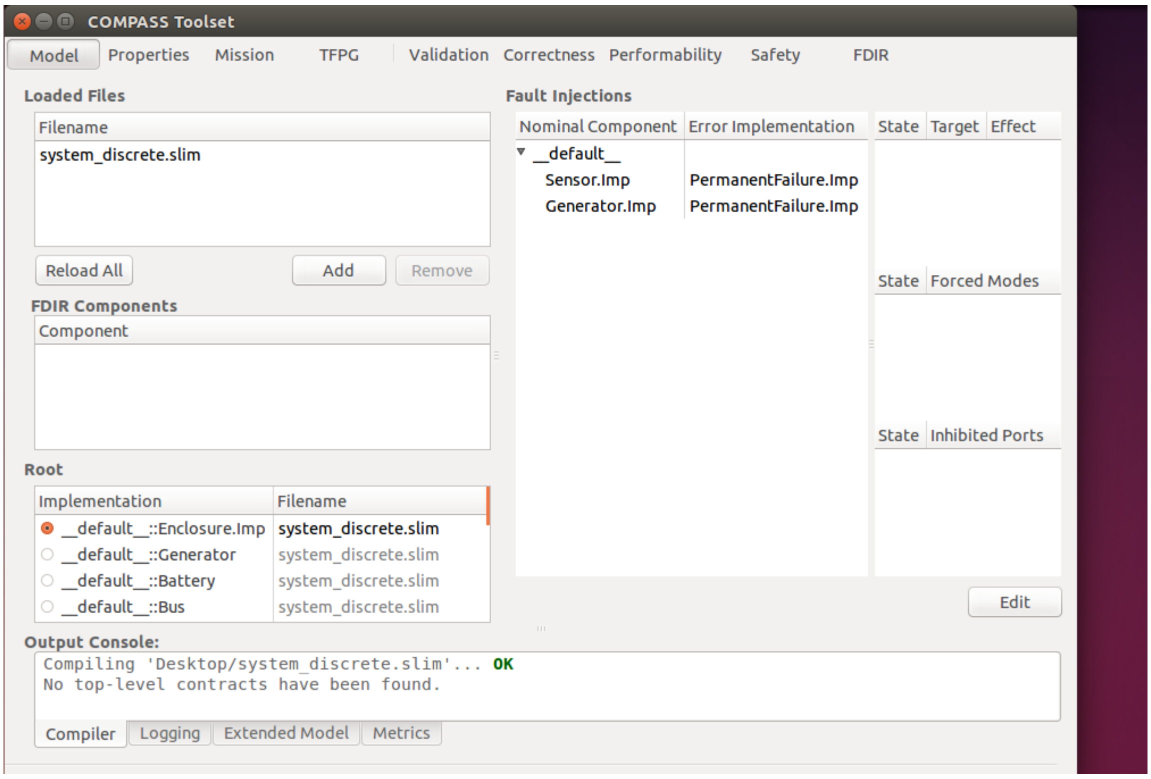
- Click the *Save* button; the new effect value is visible in the GUI by expanding again the *_default_* element and selecting `Generator.Imp`
- We can also save the changes by overwriting or saving the model with a different name as shown in Section 4.4.2

5.4 Loading and Saving Error Models and Fault Injections

Error models and fault injections can be loaded and saved along with a SLIM model, and they are stored in the same file(s) – see Section 4.4.2.

5.5 Model Extension

Model extension is the automatic process of injecting the faults – according to the fault injections specifications – into the nominal model. The outcome of model extension is an *extended model*, a SLIM model that incorporates both nominal and faulty behaviors into a single model. Model extension is performed "*behind the scenes*", in a transparent manner, every time the user wants to run an analysis on the extended model. Intuitively, the idea of model extension is that the nominal and error model are running concurrently, i.e., the state space of the extended model consists of pairs of nominal and error states, and each transition in the extended model is due to either to a nominal transition or an error transition.



Filename
system_discrete.slim

Nominal Component	Error Implementation
__default__	
Sensor.Imp	PermanentFailure.Imp
Generator.Imp	PermanentFailure.Imp

Implementation	Filename
● __default__::Enclosure.Imp	system_discrete.slim
○ __default__::Generator	system_discrete.slim
○ __default__::Battery	system_discrete.slim
○ __default__::Bus	system_discrete.slim

```
Compiling 'Desktop/system_discrete.slim'... OK
No top-level contracts have been found.
```

Figure 5.4: Fault injections for the `system_discrete` example.

5.6 Simulating the Extended Model

The extended model can be simulated in a similar manner as for the nominal model – compare Section 4.5. In the trace, faults and signals of error sub-components are marked in red. For instance, a simulation trace for the battery sensor example is shown in Figure 5.5.

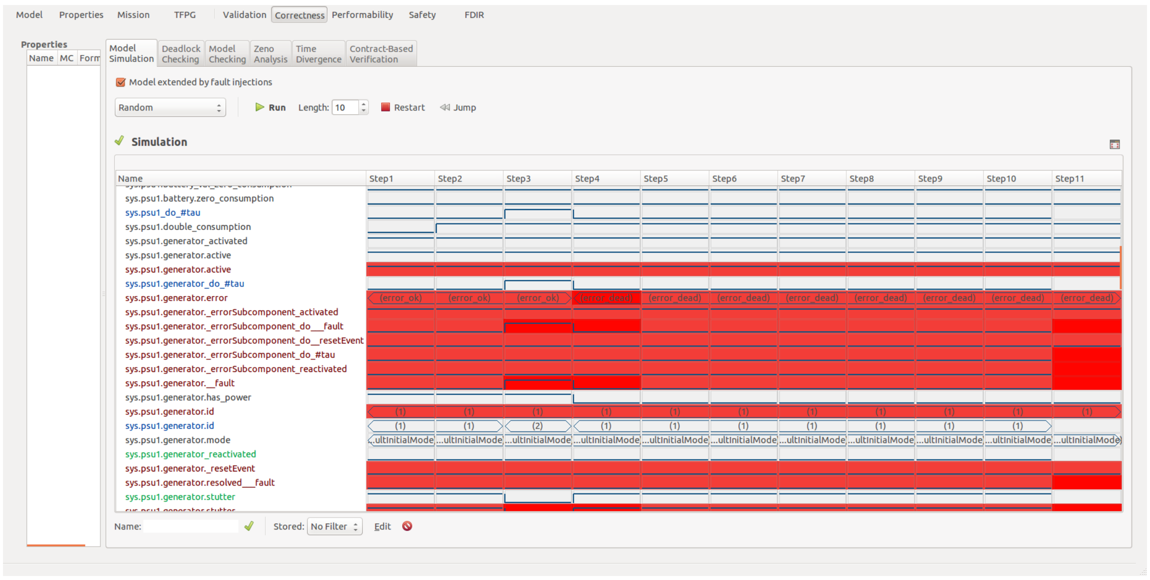
5.7 Model Checking the Extended Model

Similarly as for the nominal model, an extended model can be verified against a set of properties. Usually, properties that are satisfied nominally, may be not satisfied in presence of faults. For instance, we may expect that the battery sensor system may be not always alive, as a consequence of possible faults.

Let us see an example.

Steps

- Load the model as shown in Section 4.4.1
- Open the *File* menu and load the properties contained in file `examples/battery_sensor/system_discrete.propxml`
- Click on the *Correctness* tab and select the *Model Checking* pane
- The following properties are shown on the left *Properties* pane:



sys.psu1.battery.zero_consumption
sys.psu1_do_#tau
sys.psu1.double_consumption
sys.psu1.generator_activated
sys.psu1.generator.active
sys.psu1.generator.active
sys.psu1.generator_do_#tau
sys.psu1.generator.error
sys.psu1.generator._errorSubcomponent_activated
sys.psu1.generator._errorSubcomponent_do___fault
sys.psu1.generator._errorSubcomponent_do__resetEvent
sys.psu1.generator._errorSubcomponent_do_#tau
sys.psu1.generator._errorSubcomponent_reactivated
sys.psu1.generator.__fault
sys.psu1.generator.has_power
sys.psu1.generator.id
sys.psu1.generator.id
sys.psu1.generator.mode
sys.psu1.generator_reactivated
sys.psu1.generator._resetEvent
sys.psu1.generator.resolved___fault
sys.psu1.generator.stutter

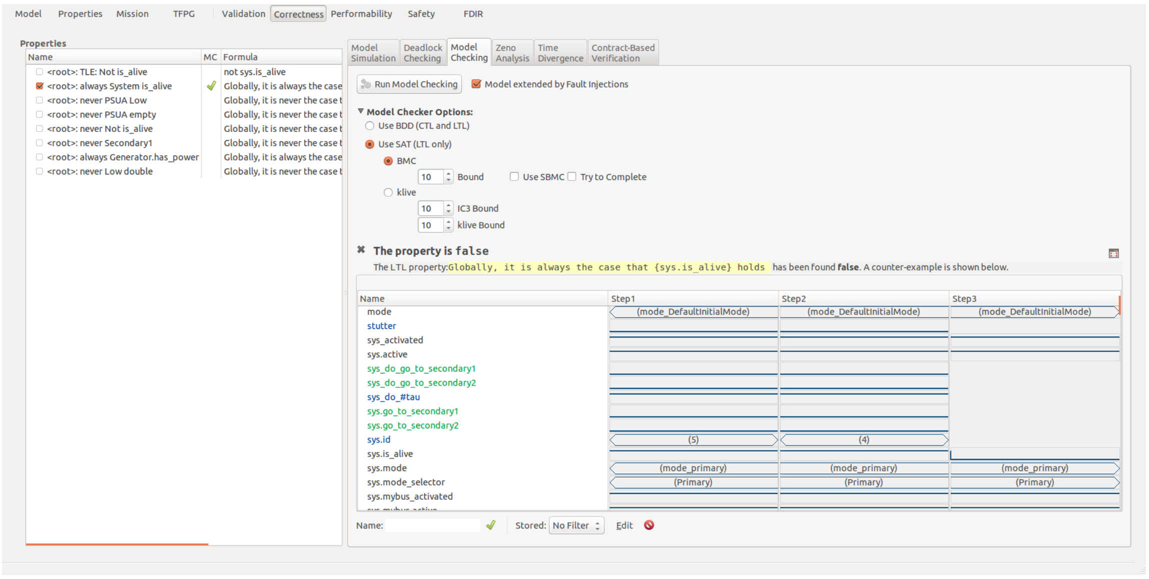
Figure 5.5: Random simulation trace created for the system_discrete example (extended model).

- `TLE: Not is_alive`
- `always System is_alive`
- `never PSUA Low`
- `never PSUA empty`
- `never Not is_alive`
- `never Secondary1`
- `always Generator.has_power`
- `never Low double`

- Select the property `always System is_alive`; this property states that the variable `is_alive` is always true
- Enable the *Model Extended by Fault Injections* checkbox
- Expand *Model Checker Options* and select the *BMC* engine
- Click the *Run Model Checking* button
- The property results to be false; a counterexample trace of length 3 is shown (see Figure 5.6) in which indeed at step 3 the element `sys.is_alive` becomes false

We can filter the counterexample by showing only the error components, by inserting "error" in the *Filtered types* of the filtering pop-up window (we refer to Section 9.4.1 of the user manual for a complete explanation) obtaining the trace shown in Figure 5.7.

As we can see, in this example the failures of the sensors cause the failure of the entire system.



Properties

Name	MC	Formula
<root>: TLE: Not is_alive		not sys.is_alive
✓ <root>: always System is_alive	✓	Globally, it is always the case
<root>: never PSUA Low		Globally, it is never the case t
<root>: never PSUA empty		Globally, it is never the case t
<root>: never Not is_alive		Globally, it is never the case t
<root>: never Secondary1		Globally, it is never the case t
<root>: always Generator.has_power		Globally, it is always the case
<root>: never Low double		Globally, it is never the case t

✓ Model extended by Fault Injections

Model Checker Options:
- Use BDD (CTL and LTL)
- [✓ Use SAT (LTL only)]
  - [✓ BMC] 10 Bound; Use SBMC; Try to Complete
  - klive: 10 IC3 Bound; 10 klive Bound

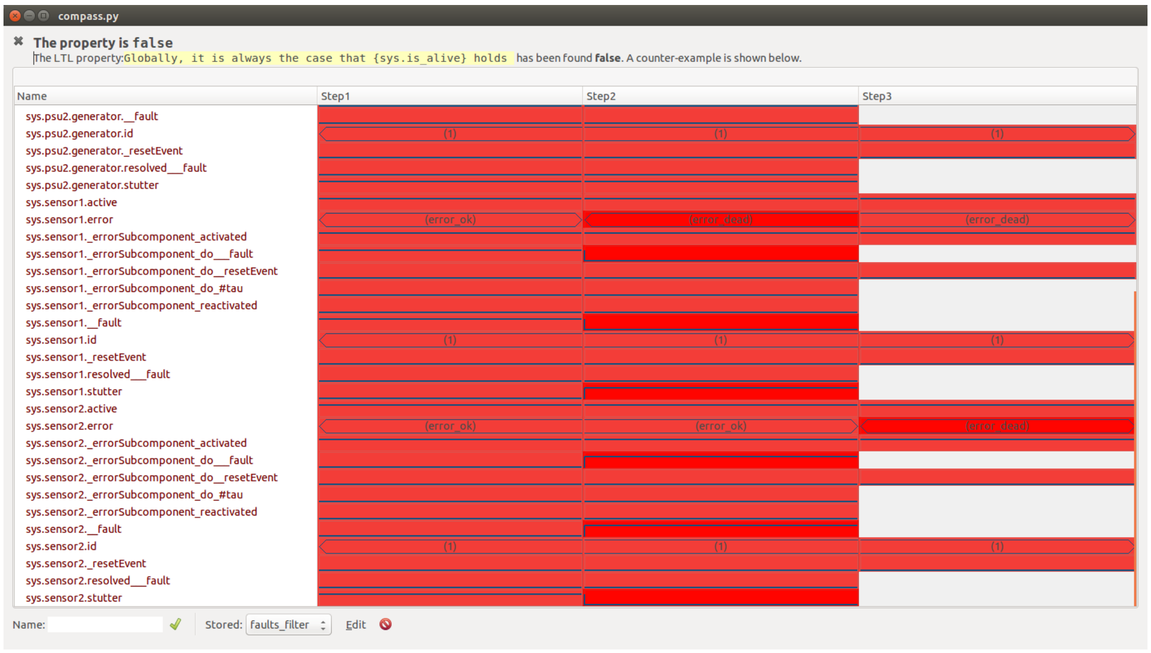
The property is false

The LTL property:`Globally, it is always the case that {sys.is_alive} holds` has been found **false**. A counter-example is shown below.

Name	Step1	Step2	Step3
mode	(mode_DefaultInitialMode)	(mode_DefaultInitialMode)	(mode_DefaultInitialMode)
stutter			
sys_activated			
sys.active			
sys_do_go_to_secondary1			
sys_do_go_to_secondary2			
sys_do_#tau			
sys.go_to_secondary1			
sys.go_to_secondary2			
sys.id	(5)	(4)	
sys.is_alive			
sys.mode	(mode_primary)	(mode_primary)	(mode_primary)
sys.mode_selector	(Primary)	(Primary)	(Primary)
sys.mybus_activated			

Name: Stored: No Filter

Figure 5.6: Counterexample trace created for the property `always System is_alive` of the `system_discrete` example (extended version).



The property is false

The LTL property:`Globally, it is always the case that {sys.is_alive} holds` has been found **false**. A counter-example is shown below.

Name	Step1	Step2	Step3
sys.psu2.generator.__fault			
sys.psu2.generator.id	(1)	(1)	(1)
sys.psu2.generator._resetEvent			
sys.psu2.generator.resolved__fault			
sys.psu2.generator.stutter			
sys.sensor1.active			
sys.sensor1.error	(error_ok)	(error_dead)	(error_dead)
sys.sensor1._errorSubcomponent_activated			
sys.sensor1._errorSubcomponent_do__fault			
sys.sensor1._errorSubcomponent_do__resetEvent			
sys.sensor1._errorSubcomponent_do_#tau			
sys.sensor1._errorSubcomponent_reactivated			
sys.sensor1.__fault			
sys.sensor1.id	(1)	(1)	(1)
sys.sensor1._resetEvent			
sys.sensor1.resolved__fault			
sys.sensor1.stutter			
sys.sensor2.active			
sys.sensor2.error	(error_ok)	(error_ok)	(error_dead)
sys.sensor2._errorSubcomponent_activated			
sys.sensor2._errorSubcomponent_do__fault			
sys.sensor2._errorSubcomponent_do__resetEvent			
sys.sensor2._errorSubcomponent_do_#tau			
sys.sensor2._errorSubcomponent_reactivated			
sys.sensor2.__fault			
sys.sensor2.id	(1)	(1)	(1)
sys.sensor2._resetEvent			
sys.sensor2.resolved__fault			
sys.sensor2.stutter			

Name: Stored: faults_filter

Figure 5.7: Example of filtered trace.

5.8 Fault Tree Analysis

Fault Tree Analysis is a traditional safety assessment technique. It is a top-down analysis that, given an undesired condition (*feared event*, also known as *top-level event*) traces it back to the set of possible causes (basic faults). The fault tree itself is a representation of the (minimal) set of causes in a tree-like structure, which uses Boolean gates (logical AND and OR) to connect tree nodes. In COMPASS, a fault tree can be generated, given a (propositional) property (representing the feared event) and the set of possible faults (those that have been specified in the error model(s) and injected by means of fault injection).

Three possible engines and different options are available; they are automatically enabled or disabled by the COMPASS toolset depending on the semantics of the model.

Let us see an example using the system_discrete example. We execute the following steps.

Steps

- Load the model as shown in Section 4.4.1
- Open the *File* menu and load the properties contained in file `examples/battery_sensor/system_discrete.propxml`
- Click on the *Safety* tab and select the *Fault Tree Generation* pane
- Only the property `TLE: Not is_alive` appears on the left (this is because it is the only propositional property)
- Select the property and click the *Generate Fault Tree* button
- A fault tree such as the one in Figure 5.8 is created. It shows the sequences of events that may cause the failure. In this example, there are:
  - two branches with a single event that causes the top-level event (in particular, an individual fault of either of the two generators is a possible cause)
  - one branch with a set of two events (linked by an AND gate) that together cause the top-level event (in particular, a failure of both sensors is a possible cause)

Each possible cause (set of faults) is known as a *minimal cut set* (MCS).

Fault trees, traditionally, are a qualitative model. However, they can be evaluated in a quantitative manner, by associating probabilities to the fault events. In COMPASS, it is possible to evaluate a fault tree by associating static probabilities to fault events, as part of the analysis. The COMPASS toolset will then automatically evaluate the probability of the intermediate nodes and of the top-level event of the fault tree.

Let us see how this can be done for the battery sensor example.

Steps

- Load the model as shown in Section 4.4.1
- Open the *File* menu and load the properties contained in file `examples/battery_sensor/system_discrete.propxml`
- Click on the *Safety* tab and select the *Fault Tree Generation* pane
- Select the property `TLE: Not is_alive` on the left
- Expand the *Model Checker Options* pane and check *Compute Probabilities*
- The bottom pane becomes editable; assign the following probabilities (see Figure 5.9):
  - `sys.psu1.generator._errorSubcomponent.__fault`: 0.2

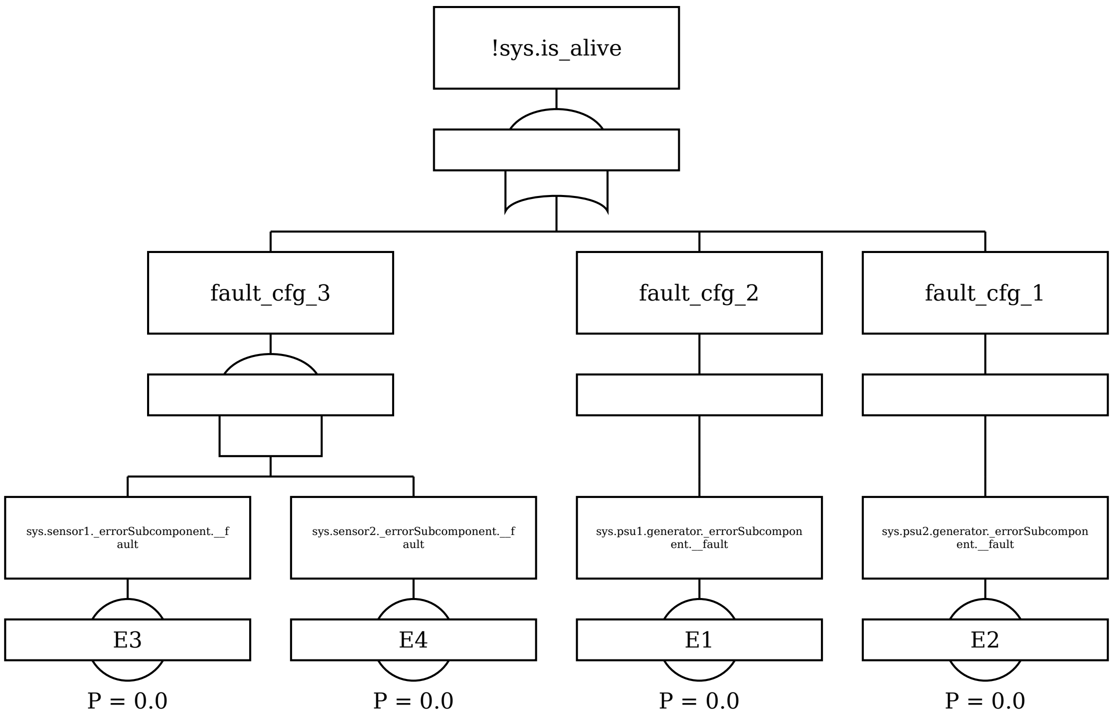

!sys.is_alive
fault_cfg_3
fault_cfg_2
fault_cfg_1
sys.sensor1._errorSubcomponent.__fault
sys.sensor2._errorSubcomponent.__fault
sys.psu1.generator._errorSubcomponent.__fault
sys.psu2.generator._errorSubcomponent.__fault
E3
E4
E1
E2
P = 0.0
P = 0.0
P = 0.0
P = 0.0

Figure 5.8: Fault Tree Generation for `system_discrete` example.

  - `sys.psu2.generator._errorSubcomponent.__fault`: 0.2
  - `sys.sensor1._errorSubcomponent.__fault`: 0.3
  - `sys.sensor2._errorSubcomponent.__fault`: 0.3

- Click the *Generate Fault Tree* button
- A fault tree such as the one in Figure 5.10 is created; it is the same fault tree as the one shown in Figure 5.8 in which probabilities are attached to each node

We can see that, under the hypotheses we have done on basic faults, the probability of the system not being alive is estimated as 0.418.

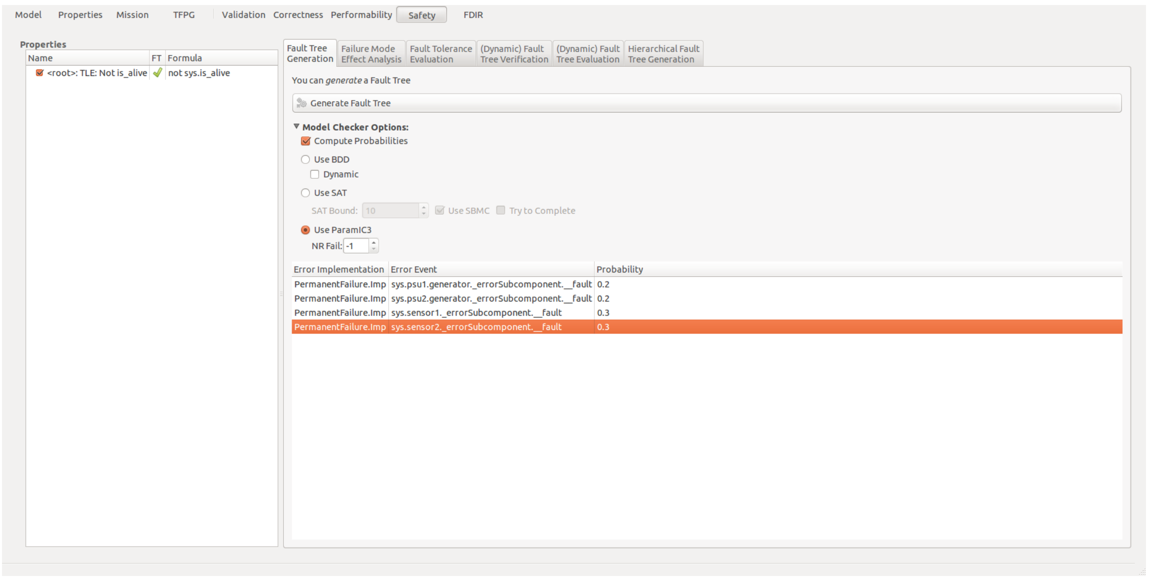
5.9 Fault Tree Evaluation

Aside from assigning static failure rates to basic events, COMPASS also allows the error rates used in error models – as described in Section 5.2 – to be used to calculate error probabilities in fault trees. This can be used to determine the overall probability of a fault tree's top level event occurring, or an arbitrary (Boolean) combination of any of the fault tree's nodes.

To see how the probability of the top level event can be calculated, take the following steps.

Steps

- Generate the fault tree as described in Section 5.8


Model
Properties
Mission
TFPG
Validation
Correctness
Performability
Safety
FDIR
Fault Tree Generation
Failure Mode Effect Analysis
Fault Tolerance Evaluation
(Dynamic) Fault Tree Verification
(Dynamic) Fault Tree Evaluation
Hierarchical Fault Tree Generation
Generate Fault Tree

Properties

Name	FT	Formula
✓ <root>: TLE: Not is_alive	✓	not sys.is_alive

You can *generate* a Fault Tree

▼ **Model Checker Options:**

- [✓] Compute Probabilities
- ○ Use BDD
  - [ ] Dynamic
- ○ Use SAT
  - SAT Bound: 10 [✓] Use SBMC [ ] Try to Complete
- ● Use ParamIC3
  - NR Fail: -1

Error Implementation	Error Event	Probability
PermanentFailure.Imp	sys.psu1.generator._errorSubcomponent.__fault	0.2
PermanentFailure.Imp	sys.psu2.generator._errorSubcomponent.__fault	0.2
PermanentFailure.Imp	sys.sensor1._errorSubcomponent.__fault	0.3
PermanentFailure.Imp	sys.sensor2._errorSubcomponent.__fault	0.3

Figure 5.9: *Fault Tree Generation* pane with probabilities enabled.

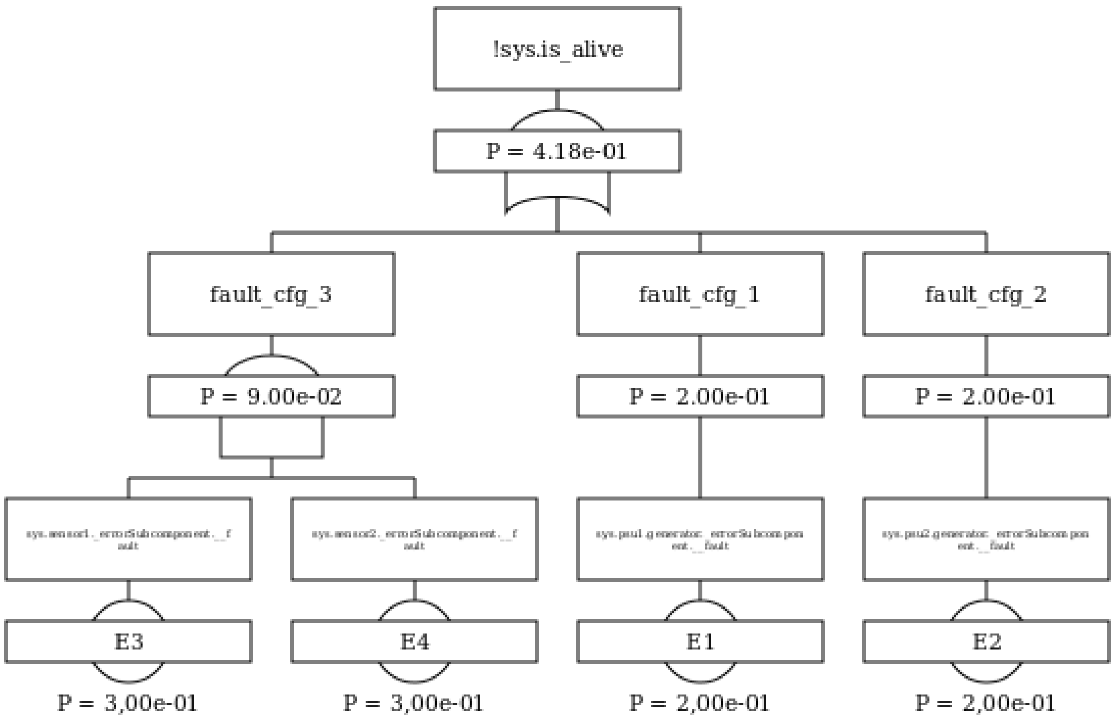

!sys.is_alive
P = 4.18e-01
fault_cfg_3
fault_cfg_1
fault_cfg_2
P = 9.00e-02
P = 2.00e-01
P = 2.00e-01
sys.sensor1._errorSubcomponent.__fault
sys.sensor2._errorSubcomponent.__fault
sys.psu1.generator._errorSubcomponent.__fault
sys.psu2.generator._errorSubcomponent.__fault
E3
E4
E1
E2
P = 3,00e-01
P = 3,00e-01
P = 2,00e-01
P = 2,00e-01

Figure 5.10: Probabilistic Fault Tree Generation for `system_discrete` example.

- Select the *(Dynamic) Fault Tree Evaluation* pane
- In the *Properties* pane on the left, ensure the property `TLE: Not is_alive` is selected
- Set the mission end time to some positive number, for example 1
- Set the severity to some positive number, for example 5

- Click the *Start Fault Tree Evaluation* button
- A fault tree such as the one in Figure 5.11 is created. It shows what the probabilities of the various events are, and their associated severity.

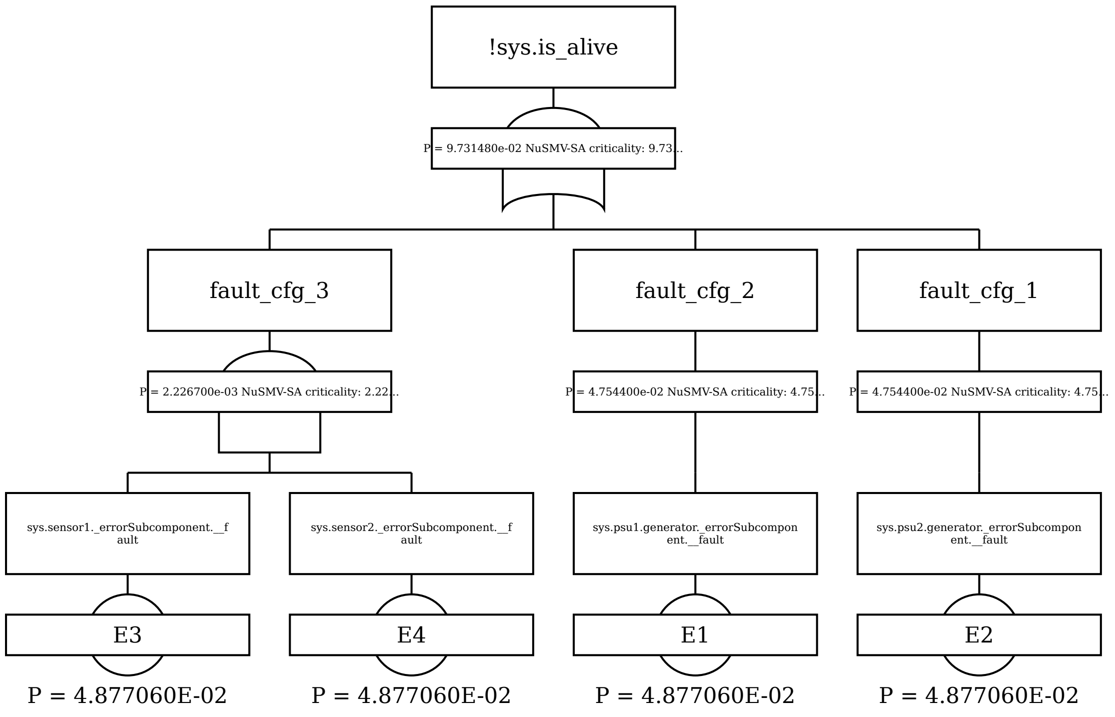

!sys.is_alive
P = 9.731480e-02 NuSMV-SA criticality: 9.73...
fault_cfg_3
fault_cfg_2
fault_cfg_1
P = 2.226700e-03 NuSMV-SA criticality: 2.22...
P = 4.754400e-02 NuSMV-SA criticality: 4.75...
P = 4.754400e-02 NuSMV-SA criticality: 4.75...
sys.sensor1._errorSubcomponent.__fault
sys.sensor2._errorSubcomponent.__fault
sys.psu1.generator._errorSubcomponent.__fault
sys.psu2.generator._errorSubcomponent.__fault
E3
E4
E1
E2
P = 4.877060E-02
P = 4.877060E-02
P = 4.877060E-02
P = 4.877060E-02

Figure 5.11: Fault Tree Evaluation for `system_discrete` example.

5.10 Fault Tree Verification

In addition to the evaluation of the fault tree, verification can be used to analyze some property over the nodes of the fault tree, determining the probability of some – optionally ordered – combination of nodes.

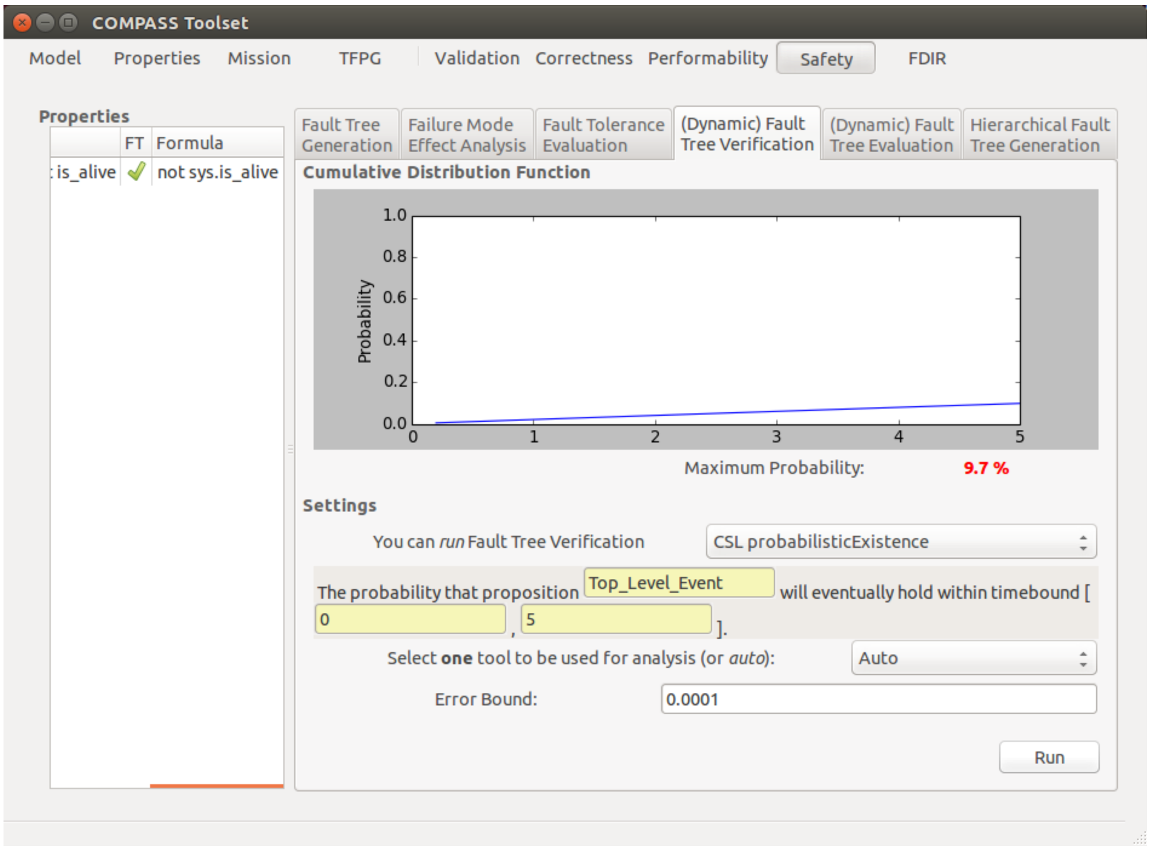
To see how such a verification of a fault tree can be performed, we take the following steps.

Steps

- Generate the fault tree as described in Section 5.8
- Select the *(Dynamic) Fault Tree Verification* pane
- In the *Properties* pane on the left, ensure the property `TLE: Not is_alive` is selected
- Choose the *CSL probabilisticExistence* pattern from the drop down list.
- In the *proposition* placeholder, enter *Top_Level_Event*
- In the *timebound* placeholders, enter 0 and 5 respectively

- Click the *Run* button
- A graph is shown showing the probability of *Top_Level_Event* eventually occurring within the time interval as specified by the placeholders.

Example output is shown in Figure 5.12.


COMPASS Toolset
Model
Properties
Mission
TFPG
Validation
Correctness
Performability
Safety
FDIR
Properties
FT
Formula
is_alive
not sys.is_alive
Fault Tree Generation
Failure Mode Effect Analysis
Fault Tolerance Evaluation
(Dynamic) Fault Tree Verification
(Dynamic) Fault Tree Evaluation
Hierarchical Fault Tree Generation
Cumulative Distribution Function
Probability
Settings
Run

Maximum Probability: 9.7 %

You can *run* Fault Tree Verification: CSL probabilisticExistence

The probability that proposition Top_Level_Event will eventually hold within timebound [0 , 5].

Select **one** tool to be used for analysis (or *auto*): Auto

Error Bound: 0.0001

Figure 5.12: Fault Tree Verification for `system_discrete` example.

5.11 Failure Mode and Effects Analysis

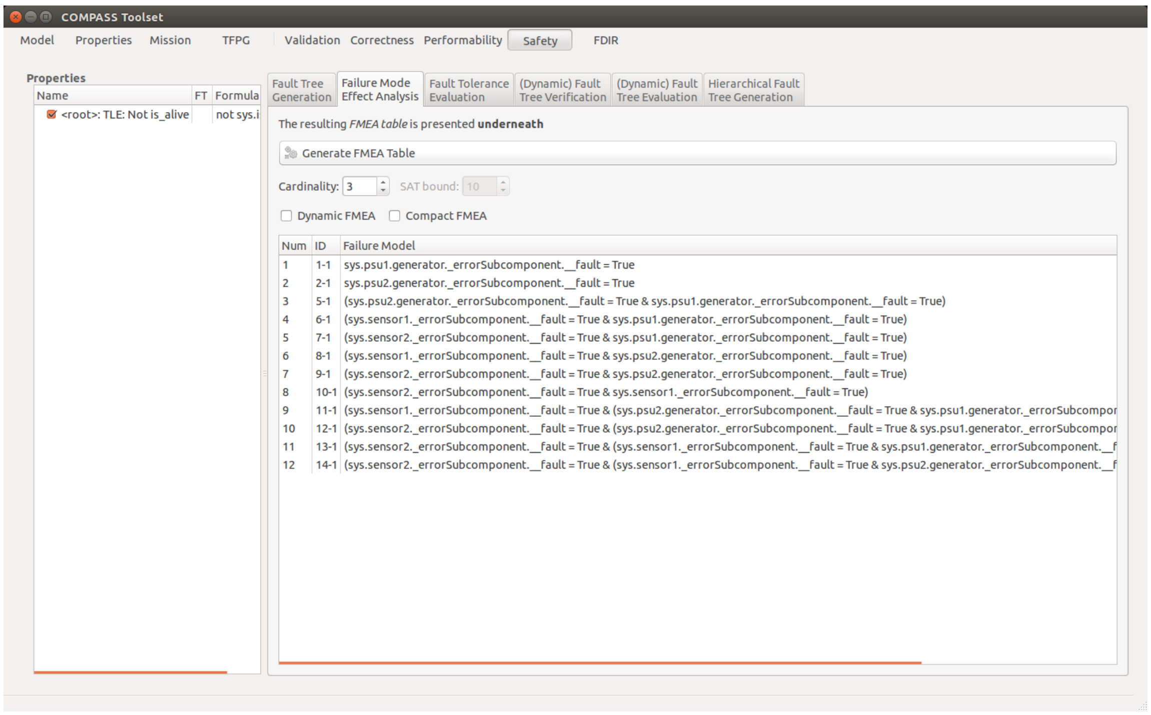
Another traditional safety assessment technique is Failure Mode and Effects Analysis (FMEA). FMEA is a bottom-up technique that, given a set of basic faults, evaluate their consequences on a set of properties (one or more). Specifically, a table (called FMEA table) is constructed, that contains one entry for each property that is being invalidated by a given set of faults. Traditionally, FMEA tables are generated for individual faults (FMEA table with cardinality one), however in COMPASS it is possible to generate FMEA tables for higher cardinality.

Let us see an example with the `system_discrete` model. We execute the following steps.

Steps

- Load the model as shown in Section 4.4.1

- Open the *File* menu and load the properties contained in file `examples/battery_sensor/system_discrete.propxml`
- Click on the *Safety* tab and select the *Fault Tree Generation* pane
- Select the property `TLE: Not is_alive` on the left
- Set the value of *Cardinality* to 3
- Click the *Generate FMEA Table* button, which fills the table below with the results of the computation. As shown in Figure 5.13, FMEA proved that possible causes of the event "`not sys.is_alive`" are:
  1. two combinations of cardinality one (`sys.psu1.generator._errorSubcomponent.__fault` and `sys.psu2.generator._errorSubcomponent.__fault`)
  2. six combinations of cardinality two (rows having ID from `5-1` ID `10-1`)
  3. four combinations of cardinality three (rows having ID from `11-1` to `14-1`)


COMPASS Toolset
Model
Properties
Mission
TFPG
Validation
Correctness
Performability
Safety
FDIR
Properties
Name
FT
Formula
<root>: TLE: Not is_alive
not sys.i
Fault Tree Generation
Failure Mode Effect Analysis
Fault Tolerance Evaluation
(Dynamic) Fault Tree Verification
(Dynamic) Fault Tree Evaluation
Hierarchical Fault Tree Generation
Generate FMEA Table

The resulting *FMEA table* is presented **underneath**

Cardinality: 3 SAT bound: 10

☐ Dynamic FMEA ☐ Compact FMEA

Num	ID	Failure Model
1	1-1	sys.psu1.generator._errorSubcomponent.__fault = True
2	2-1	sys.psu2.generator._errorSubcomponent.__fault = True
3	5-1	(sys.psu2.generator._errorSubcomponent.__fault = True & sys.psu1.generator._errorSubcomponent.__fault = True)
4	6-1	(sys.sensor1._errorSubcomponent.__fault = True & sys.psu1.generator._errorSubcomponent.__fault = True)
5	7-1	(sys.sensor2._errorSubcomponent.__fault = True & sys.psu1.generator._errorSubcomponent.__fault = True)
6	8-1	(sys.sensor1._errorSubcomponent.__fault = True & sys.psu2.generator._errorSubcomponent.__fault = True)
7	9-1	(sys.sensor2._errorSubcomponent.__fault = True & sys.psu2.generator._errorSubcomponent.__fault = True)
8	10-1	(sys.sensor2._errorSubcomponent.__fault = True & sys.sensor1._errorSubcomponent.__fault = True)
9	11-1	(sys.sensor1._errorSubcomponent.__fault = True & (sys.psu2.generator._errorSubcomponent.__fault = True & sys.psu1.generator._errorSubcompor
10	12-1	(sys.sensor2._errorSubcomponent.__fault = True & (sys.psu2.generator._errorSubcomponent.__fault = True & sys.psu1.generator._errorSubcompor
11	13-1	(sys.sensor2._errorSubcomponent.__fault = True & (sys.sensor1._errorSubcomponent.__fault = True & sys.psu1.generator._errorSubcomponent.__f
12	14-1	(sys.sensor2._errorSubcomponent.__fault = True & (sys.sensor1._errorSubcomponent.__fault = True & sys.psu2.generator._errorSubcomponent.__f

Figure 5.13: FMEA with cardinality 3 for `system_discrete` example.

More details on the different options of FMEA (*Dynamic FMEA* and *Compact FMEA*) can be found in Section 9.6.4 of the user manual.

5.12 Modeling Fault Propagation: TFPGs

The COMPASS toolset allows to perform many analyses using Times Failure Propagation Graphs (TFPG). TFPGs are a powerful formalism to model and analyze failure propagation behavior in a model. In other words, a TFPG is an abstract view of a system model that represent only the failure propagation information: basic faults and their effects (called *discrepancies*). Faults and effects are represented as nodes in a graph – an edge connecting different node indicates a failure propagation. Moreover, an edge may be labeled with timing and mode information (time needed for the propagation to take place, and system modes that enable the propagation). TFPGs can be used for diagnostic or prognostic purposes.

More in detail, a TFPG is a labeled directed graph where nodes represent either failure 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 $[t_{\min}, t_{\max}]$. Edges in the graph model can be activated or deactivated depending on a set of possible operational modes of the system (*activation modes*); this allows to represent failure propagation in multi-mode (switching) systems.

We notice that discrepancies may be either monitored or non-monitored (observable or not observable)[1].

It is possible to relate a TFPG with a given system model, in order to analyze whether the TFPG is indeed an abstraction of the model (i.e., it contains at least as many behaviors as the system model). Moreover, it is possible to automatically synthesize a TFPG, given a system model, a set of faults and the specification of the discrepancies (in terms of the system model). For running these analyses, we need to specify some *TFPG SLIM associations*, which define TFPG elements (failure modes and discrepancies) in terms of SLIM propositional expressions (predicating on the system model). 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 discrepancies in the TFPG to SLIM expressions
3. Associate TFPG modes to SLIM expressions

We refer to section 9.3 of the user-manual for a more exhaustive explanation about TFPGs and TFPG analyses.

5.12.1 Editing TFPGs

We now show how a TFPG can be edited. We first show how it can be edited from scratch.

As an example, we assume we want to build a TFPG for the battery sensor example, with the following failure modes:

- `Gen1_off`
- `Gen2_off`
- `Sens1_off`
- `Sens2_off`

[1]We will discuss this aspect later in Section 7 when we introduce observability and FDIR analyses.

the following discrepancies:

- `System_Dead`
- `G1_DEAD`
- `B1_LOW`
- `B1_DEAD`
- `S1_WO`
- `G2_DEAD`
- `B2_LOW`
- `B2_DEAD`
- `S2_WO`

and the following system modes:

- `Primary`
- `Secondary1`
- `Secondary2`

Moreover, we assume that propagations take the following propagation times and can take place in the following system modes:

- from `Gen1_off` to `G1_DEAD` with propagation time $[0.0, 0.0]$
  in modes: Primary,Secondary1,Secondary2
- from `G1_DEAD` to `B1_LOW` with propagation time $[0.0, 3.0]$
  in modes: Primary,Secondary1
- from `B1_LOW` to `B1_DEAD` with propagation time $[1.0, 2.0]$
  in modes: Primary,Secondary1
- from `B1_DEAD` to `S1_WO` with propagation time $[0.0, 1.0]$
  in modes: Primary,Secondary1
- from `S1_WO` to `System_Dead` with propagation time $[0.0, 0.0]$
  in modes: Primary,Secondary1,Secondary2
- from `Gen2_off` to `G2_DEAD` with propagation time $[0.0, 0.0]$
  in modes: Primary,Secondary1,Secondary2
- from `G2_DEAD` to `B2_LOW` with propagation time $[0.0, 3.0]$
  in modes: Primary,Secondary2

- from `B2_LOW` to `B2_DEAD` with propagation time $[1.0, 2.0]$
  in modes: Primary,Secondary2
- from `B2_DEAD` to `S2_WO` with propagation time $[0.0, 1.0]$
  in modes: Primary,Secondary2
- from `S2_WO` to `System_Dead` with propagation time $[0.0, 0.0]$
  in modes: Primary,Secondary1,Secondary2
- from `B1_DEAD` to `S2_WO` with propagation time $[0.0, 1.0]$
  in modes: Secondary1
- from `B2_DEAD` to `S1_WO` with propagation time $[0.0, 1.0]$
  in modes: Secondary2
- from `Sens1_off` to `S1_WO` with propagation time $[0.0, 1.0]$
  in modes: Primary,Secondary1,Secondary2
- from `Sens2_off` to `S2_WO` with propagation time $[0.0, 1.0]$
  in modes: Primary,Secondary1,Secondary2

The idea is that, as in the battery sensor system presented so far, in the normal operating mode (*Primary*) each sensor is powered by its own battery; however, two other configurations are possible: *Secondary1* and *Secondary2*. These configurations are named according to the number of the battery actually providing the energy for the sensors. Two types of failure are possible: *Generator* and *Sensor*. If the generator fails, the battery starts discharging at a constant rate given by the number of sensors that are connected to it. When the battery becomes exhausted, the sensors attached to it stop working. Discrepancies `S1_WO` and `S2_WO` model the fact that the respective sensor has stopped working.

Discrepancies can be monitored or not. In this example, we decided to monitor only these discrepancies: *B1_LOW*, *B2_LOW* (representing a low charge of the corresponding battery) and *System_Dead* (representing a system failure, i.e., the system being not alive). The choice of which discrepancies are to be monitored might depend (in practice) on the availability of suitable sensors. The non-determinism on the propagation time of the failure is mainly given by the fact that we do not know the charge level of the battery until we get to the critical level *(BX_LOW)*. Additionally, we allow for a small non-determinism before activating the discrepancy indicating a wrong reading *(SX_WO)*. Note that there is uncertainty on the propagation time between *BX_LOW* and *BX_DEAD*. This is motivated by the fact that the depletion of the battery will take more time if we are in the *Primary* mode rather than in the *Secondary* mode.

We can create the TFPG in the following way.

Steps

- Load the model as shown in Section 4.4.1
- Click on the *TFPG* tab
- In the bottom-left part of the window, click the *New* button

- Choose a name for the file to be created (e.g `example_tfpg.tfpg`)
- Click the *Open* button and then *OK*; a text editor pops-up
- Insert the data as shown in Figure 5.14
- Save and close the text editor
- The graphical representation of the TFPG is shown in the toolset (see Figure 5.15); on the left side it is possible to see the TFPG named `Example-TFPG`

```
NAME Example-TFPG
INFINITY_SEMANTICS_CLOSED
FM Gen1_off
FM Gen2_off
FM Sens1_off
FM Sens2_off
AND System_Dead MONITORED
OR G1_DEAD
OR B1_LOW MONITORED
OR B1_DEAD
OR S1_WO
OR G2_DEAD
OR B2_LOW MONITORED
OR B2_DEAD
OR S2_WO
MODES Primary,Secondary1,Secondary2
EDGE EDGE1 Gen1_off G1_DEAD 0.0 0.0 (Primary,Secondary1,Secondary2)
EDGE EDGE2 G1_DEAD B1_LOW 0.0 3.0 (Primary,Secondary1)
EDGE EDGE3 B1_LOW B1_DEAD 1.0 2.0 (Primary,Secondary1)
EDGE EDGE4 B1_DEAD S1_WO 0.0 1.0 (Primary,Secondary1)
EDGE EDGE5 S1_WO System_Dead 0.0 0.0 (Primary,Secondary1,Secondary2)
EDGE EDGE6 Gen2_off G2_DEAD 0.0 0.0 (Primary,Secondary1,Secondary2)
EDGE EDGE7 G2_DEAD B2_LOW 0.0 3.0 (Primary,Secondary2)
EDGE EDGE8 B2_LOW B2_DEAD 1.0 2.0 (Primary,Secondary2)
EDGE EDGE9 B2_DEAD S2_WO 0.0 1.0 (Primary,Secondary2)
EDGE EDGE10 S2_WO System_Dead 0.0 0.0 (Primary,Secondary1,Secondary2)
EDGE EDGE11 B1_DEAD S2_WO 0.0 1.0 (Secondary1)
EDGE EDGE12 B2_DEAD S1_WO 0.0 1.0 (Secondary2)
EDGE EDGE13 Sens1_off S1_WO 0.0 1.0 (Primary,Secondary1,Secondary2)
EDGE EDGE14 Sens2_off S2_WO 0.0 1.0 (Primary,Secondary1,Secondary2)
```

Figure 5.14: Example of TFPG written in textual format.

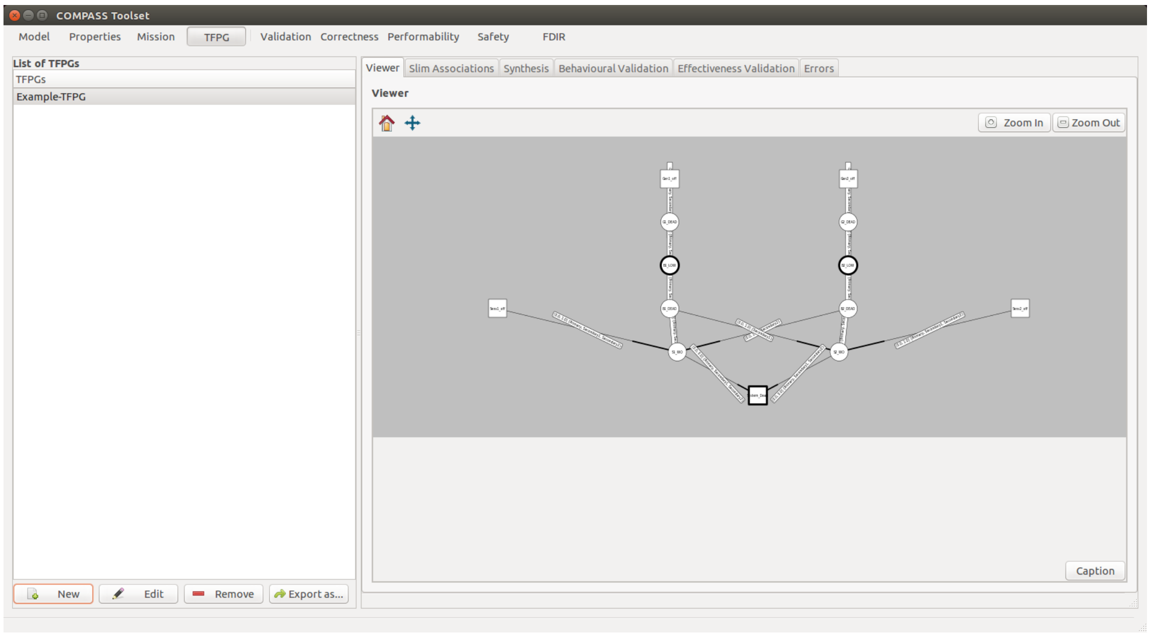


Figure 5.15: Example of a TFPG for the `system_discrete` example.

5.12.2 Loading and Saving TFPGs

In addition to creating a new TFPG as shown in the previous section, it is possible to load and modify an existing TFPG, and to save a TFPG. For instance, we can load the xml representation of the previous TFPG as follows.

Steps

- Load the model as shown in Section 4.4.1
- Click on the *TFPG* tab
- Click on the *File* menu, choose the item *Load TFPG...* and select the TFPG `examples/battery_sensor/system.txml`; a graphical representation of the TFPG is shown in the *Viewer* tab (see Figure 5.15)

Now, we can edit the loaded TFPG or save it with a different name as follows.

Steps

- Load the TFPG as shown previously
- Click the *File* menu
- Select the item *Save TFPG As...*
- Insert `copy_of_system_discrete.txml` as name and save it to Desktop
- Click the *Save* button; a file named `copy_of_system_discrete.txml` has been created in the Desktop

- In the GUI, select the TFPG `Sensor-Generator` and click the *Edit* button; an external editor opens
- Change the name of the TFPG (modify the first line from "NAME Sensor-Generator" to "NAME Sensor-Generator-Old")
- Delete the last line
  "EDGE EDGE14 Sens2_off S2_WO 0.0 1.0 (Primary,Secondary1,Secondary2)"
- Save and close the external editor; the updated TFPG (i.e., an edge is now missing) is shown in the *Viewer* pane
- Click the *File* menu
- Select the item *Load TFPG...* and select the file `copy_of_system_discrete.txml` previously saved to the Desktop; the unmodified version of the TFPG is shown in the *Viewer* tab. In particular (see Figure 5.16), on the left side the two TFPGs appear (`Sensor-Generator-Old` and `Sensor-Generator`)

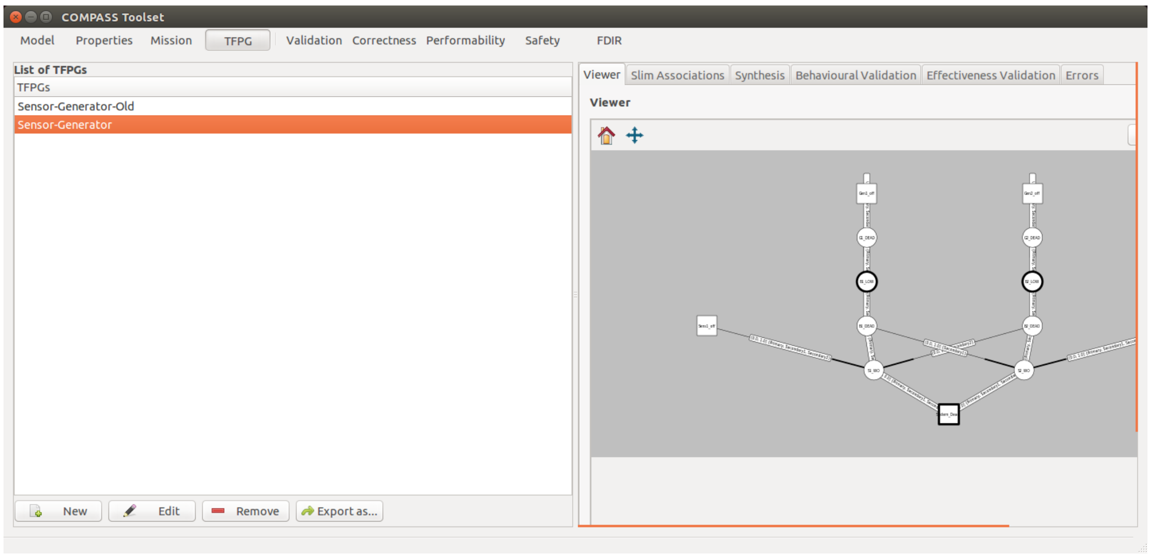


Figure 5.16: Two different loaded TFPGs for the `system_discrete` example.

5.12.3 Editing TFPG Associations

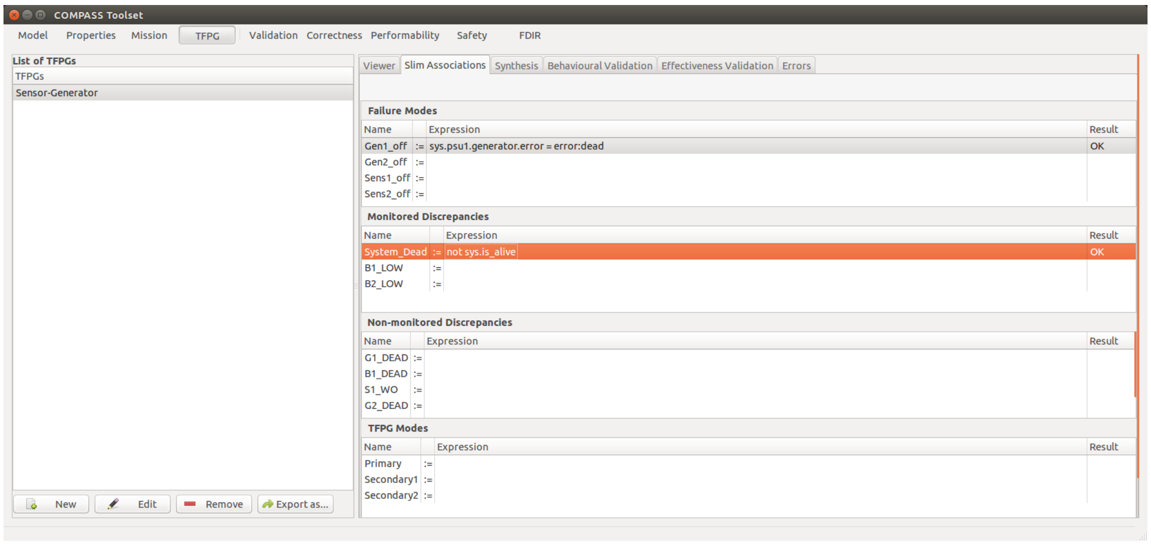
TFPG associations provide the link between a TFPG and a system model. They are used to run TFPG analyses, such as behavioral validation and synthesis - that are illustrated later on in this section.

TFPG associations may be edited using the COMPASS toolset as follows.

Steps

- Load the model as shown in Section 4.4.1

- Click on the *TFPG* tab
- Click on the *File* menu, select the item *Load TFPG...* and select the TFPG `examples/battery_sensor/system.txml`; a graphical representation of the TFPG is shown in the *Viewer* tab
- Click on the *Slim Associations* tab; the names of the different TFPG elements are divided depending on their category (*Failure Modes*, *Monitored Discrepancies*, *Non-monitored Discrepancies* and *TFPG Modes*); now it is possible to associate some SLIM expressions
- In the row of the failure mode `Gen1_off`, select the following expression from the drop-down menu: `sys.psu1.generator.error = error:dead`. In this way, we associate to the failure mode `Gen1_off` the expression corresponding to the generator being in `dead` state. An `OK` in the *Result* columns confirms that the expression we have inserted is correct.
- In the row of the monitored discrepancy `System_Dead`, type the following expression: `not sys.is-alive`. An error message is shown and in the *Result* column the value `ERROR` appears. Correct the expression to `not sys.is_alive` and the result `OK` is shown (see Figure 5.17).



List of TFPGs

TFPGs
Sensor-Generator

Failure Modes

Name		Expression	Result
Gen1_off	:=	sys.psu1.generator.error = error:dead	OK
Gen2_off	:=		
Sens1_off	:=		
Sens2_off	:=		

Monitored Discrepancies

Name		Expression	Result
System_Dead	:=	not sys.is_alive	OK
B1_LOW	:=		
B2_LOW	:=		

Non-monitored Discrepancies

Name		Expression	Result
G1_DEAD	:=		
B1_DEAD	:=		
S1_WO	:=		
G2_DEAD	:=		

TFPG Modes

Name		Expression	Result
Primary	:=		
Secondary1	:=		
Secondary2	:=		

Figure 5.17: Adding TFPG SLIM associations for the `system_discrete` example.

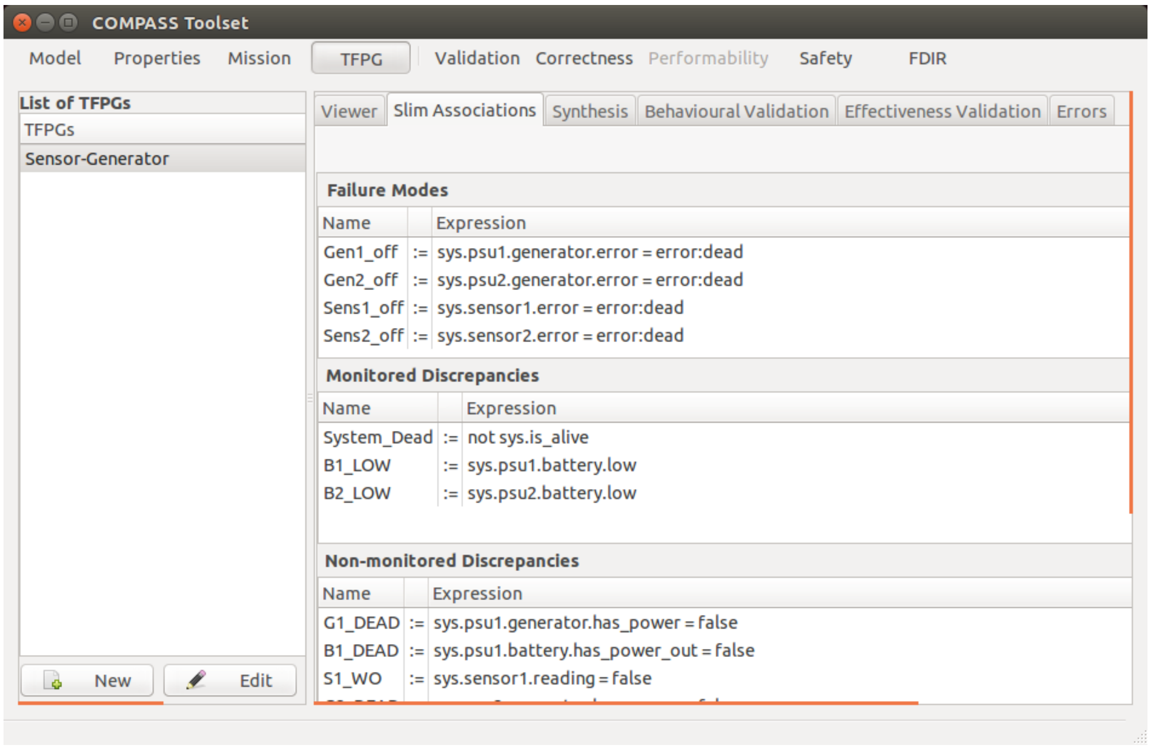
5.12.4 Loading and Saving TFPG Associations

Similarly to TFPGs, TFPG associations may be loaded and saved. In order to load associations for the `system_discrete` example, we can use the following steps.

Steps

- Load the model as shown in Section 4.4.1

- Click on the *TFPG* tab
- Click on the *File* menu, select the item *Load TFPG...* and select the TFPG `examples/battery_sensor/system.txml`; a graphical representation of the TFPG is shown in the *Viewer* tab
- Click on the *Slim Associations* tab
- Click the *Load* button in the bottom part and select the file `examples/battery_sensor/system.axml`; the pane is filled with the SLIM associations as shown in Figure 5.18



Failure Modes

Name		Expression
Gen1_off	:=	sys.psu1.generator.error = error:dead
Gen2_off	:=	sys.psu2.generator.error = error:dead
Sens1_off	:=	sys.sensor1.error = error:dead
Sens2_off	:=	sys.sensor2.error = error:dead

Monitored Discrepancies

Name		Expression
System_Dead	:=	not sys.is_alive
B1_LOW	:=	sys.psu1.battery.low
B2_LOW	:=	sys.psu2.battery.low

Non-monitored Discrepancies

Name		Expression
G1_DEAD	:=	sys.psu1.generator.has_power = false
B1_DEAD	:=	sys.psu1.battery.has_power_out = false
S1_WO	:=	sys.sensor1.reading = false

Figure 5.18: TFPG SLIM associations for the `system_discrete` example.

The *Save* button located in the bottom part of the GUI, allows to save the associations in the same way as shown for TFPGs.

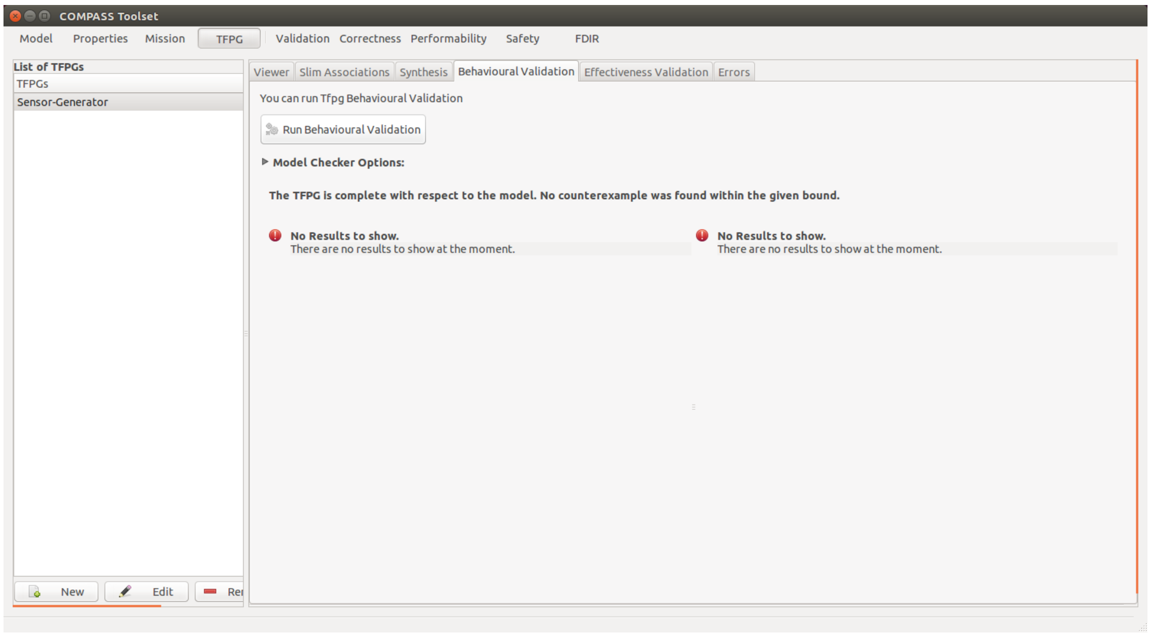
5.12.5 TFPG Behavioral Validation

Behavioral validation has the purpose of checking whether a TFPG is a complete abstraction of a system model, that is, that all behaviors of the system model are indeed specified in the TFPG. Behavioral Validation for the `system_discrete` example can be carried out through the following steps.

Steps

- Load the model as shown in Section 4.4.1

- Click on the *TFPG* tab
- Click on the *File* menu, select the item *Load TFPG...* and select the TFPG `examples/battery_sensor/system.txml`; a graphical representation of the TFPG is shown in the *Viewer* tab
- Click on the *Slim Associations* tab
- Click the *Load* button in the bottom part and select the file `examples/battery_sensor/system.axml`; the pane is filled with the SLIM associations.
- Click on the *Tfpg Behavioral Validation* tab
- Click the *Run Behavioral Validation* button
- A message confirms that the TFPG is complete with respect to the model (see Figure 5.19)



You can run Tfpg Behavioural Validation

The TFPG is complete with respect to the model. No counterexample was found within the given bound.

No Results to show.
There are no results to show at the moment.

No Results to show.
There are no results to show at the moment.

Figure 5.19: TFPG Behavioral Validation for the `system_discrete` example.

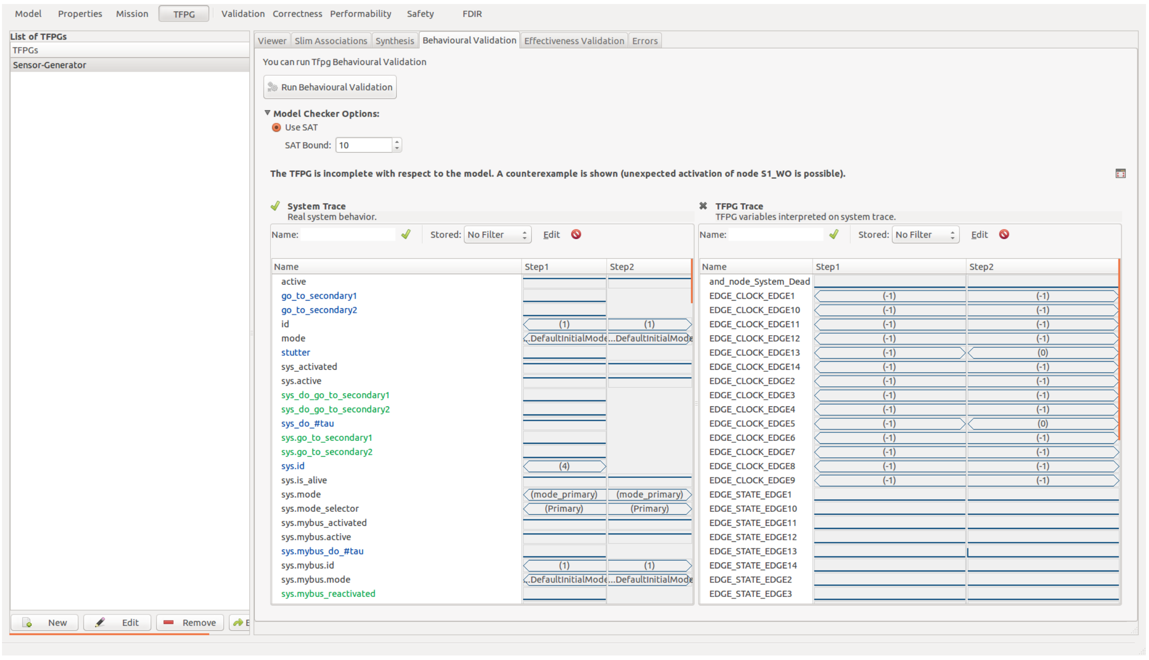
In case the TFPG is not complete with respect to the system model, a message appears along with the counterexample, which consists of a pair of traces: a system trace and the corresponding view of the system trace for the TFPG.

As an example, let us edit the TFPG definition as follows.

Steps

- Load the model as shown in Section 4.4.1
- Click on the *TFPG* tab

- Click on the *File* menu, select the item *Load TFPG...* and select the TFPG `examples/battery_sensor/system.txml`; a graphical representation of the TFPG is shown in the *Viewer* tab
- Click the *Edit* button; an external editor opens.
- Replace the destination node (`S1_WO`) of the edge connecting `Sens1_Off` with `S2_WO`; close and save the external editor. The modified graphical representation of the TFPG is shown.
- Click on the *Slim Associations* tab
- Click the *Load* button in the bottom part and select the file `examples/battery_sensor/system.axml`; the pane is filled with the SLIM associations.
- Click on the *Behavioral Validation* tab
- Click the *Run Behavioral Validation* button
- The TFPG is indeed incomplete with respect to the system model (see Figure 5.20) and the reason, as expected, is that the node `S1_WO` can now be activated without the failure of the sensor `sensor1`.



You can run Tfpg Behavioural Validation

Model Checker Options:
[✓ Use SAT]
SAT Bound: 10

The TFPG is incomplete with respect to the model. A counterexample is shown (unexpected activation of node S1_WO is possible).

System Trace
Real system behavior.
Name: Stored: No Filter Edit

Name	Step1	Step2
active		
go_to_secondary1		
go_to_secondary2		
id	(1)	(1)
mode	..DefaultInitialMode	...DefaultInitialMode
stutter		
sys_activated		
sys.active		
sys_do_go_to_secondary1		
sys_do_go_to_secondary2		
sys_do_#tau		
sys.go_to_secondary1		
sys.go_to_secondary2		
sys.id	(4)	
sys.is_alive		
sys.mode	(mode_primary)	(mode_primary)
sys.mode_selector	(Primary)	(Primary)
sys.mybus_activated		
sys.mybus.active		
sys.mybus_do_#tau		
sys.mybus.id	(1)	(1)
sys.mybus.mode	..DefaultInitialMode	...DefaultInitialMode
sys.mybus_reactivated		

TFPG Trace
TFPG variables interpreted on system trace.
Name: Stored: No Filter Edit

Name	Step1	Step2
and_node_System_Dead		
EDGE_CLOCK_EDGE1	(-1)	(-1)
EDGE_CLOCK_EDGE10	(-1)	(-1)
EDGE_CLOCK_EDGE11	(-1)	(-1)
EDGE_CLOCK_EDGE12	(-1)	(-1)
EDGE_CLOCK_EDGE13	(-1)	(0)
EDGE_CLOCK_EDGE14	(-1)	(-1)
EDGE_CLOCK_EDGE2	(-1)	(-1)
EDGE_CLOCK_EDGE3	(-1)	(-1)
EDGE_CLOCK_EDGE4	(-1)	(-1)
EDGE_CLOCK_EDGE5	(-1)	(0)
EDGE_CLOCK_EDGE6	(-1)	(-1)
EDGE_CLOCK_EDGE7	(-1)	(-1)
EDGE_CLOCK_EDGE8	(-1)	(-1)
EDGE_CLOCK_EDGE9	(-1)	(-1)
EDGE_STATE_EDGE1		
EDGE_STATE_EDGE10		
EDGE_STATE_EDGE11		
EDGE_STATE_EDGE12		
EDGE_STATE_EDGE13		
EDGE_STATE_EDGE14		
EDGE_STATE_EDGE2		
EDGE_STATE_EDGE3		

Figure 5.20: TFPG Behavioral Validation for the `system_discrete` example (incomplete result).

5.12.6 TFPG Synthesis

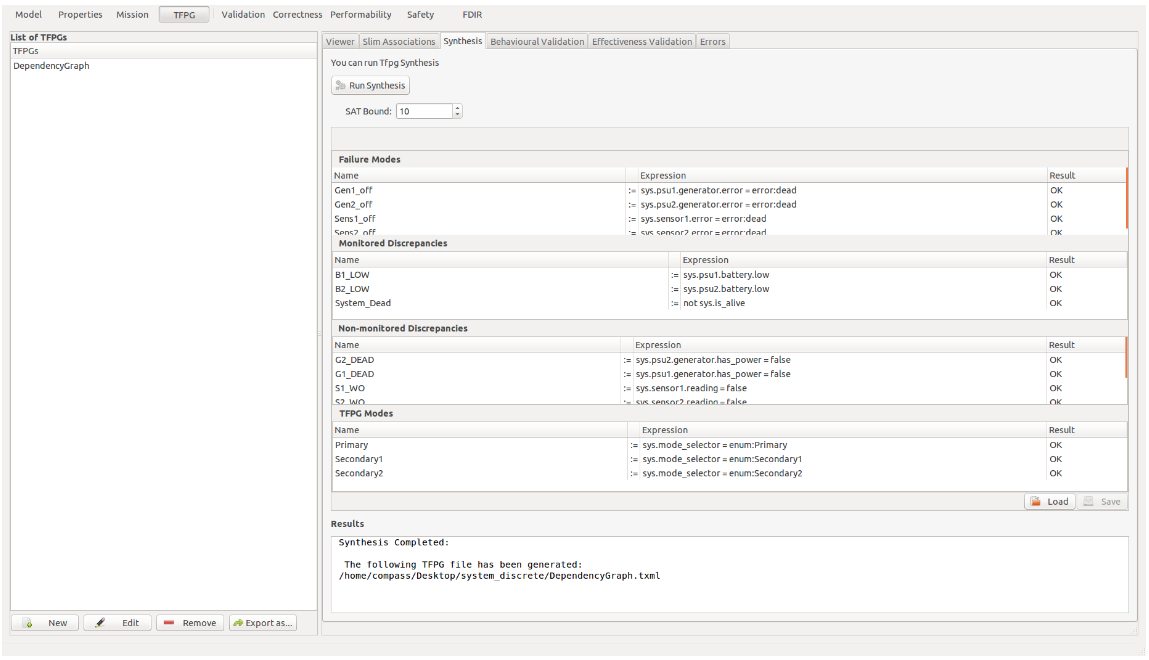
Using TFPG synthesis, it is possible to automatically synthesize a complete TFPG starting from a system model and a set of TFPG associations.

As ax example, let us assume that we want to automatically synthesize a TFPG, given the `system_discrete` example, and the same TFPG associations shown in Section 5.12.3.

We execute the following steps.

Steps

- Load the model as shown in Section 4.4.1
- Click on the *TFPG* tab
- Click on the *Synthesis* tab
- Click the *Load* button in the bottom part and select the file `examples/battery_sensor/system.axml`; the pane is filled with the SLIM associations.
- Click the *Run Synthesis* button
- A message confirms that the analysis is completed, and a TFPG named `DependencyGraph` is shown on the left side (see Figure 5.21); clicking on it will load the graphical representation in the *Viewer* tab



You can run Tfpg Synthesis

SAT Bound: 10

Failure Modes

Name		Expression	Result
Gen1_off	:=	sys.psu1.generator.error = error:dead	OK
Gen2_off	:=	sys.psu2.generator.error = error:dead	OK
Sens1_off	:=	sys.sensor1.error = error:dead	OK
Sens2_off	:=	sys.sensor2.error = error:dead	OK

Monitored Discrepancies

Name		Expression	Result
B1_LOW	:=	sys.psu1.battery.low	OK
B2_LOW	:=	sys.psu2.battery.low	OK
System_Dead	:=	not sys.is_alive	OK

Non-monitored Discrepancies

Name		Expression	Result
G2_DEAD	:=	sys.psu2.generator.has_power = false	OK
G1_DEAD	:=	sys.psu1.generator.has_power = false	OK
S1_WO	:=	sys.sensor1.reading = false	OK
S2_WO	:=	sys.sensor2.reading = false	OK

TFPG Modes

Name		Expression	Result
Primary	:=	sys.mode_selector = enum:Primary	OK
Secondary1	:=	sys.mode_selector = enum:Secondary1	OK
Secondary2	:=	sys.mode_selector = enum:Secondary2	OK

Results

```
Synthesis Completed:

 The following TFPG file has been generated:
/home/compass/Desktop/system_discrete/DependencyGraph.txml
```

Figure 5.21: TFPG Synthesis for the `system_discrete` example.

We remark that currently, only the TFPG graph is synthesized. The edge timings are set to $[0, +\infty]$ and the edges labeled with all the modes, which guarantees TFPG completeness. The TFPG timings and modes can then be manually edited, if needed.

Chapter 6

Probabilistic Verification

Aside from *qualitative* or *timed* model checking as described in Chapter 4, COMPASS also provides the capability for *quantitative* model checking, in the form of probabilistic verification. If the model contains error events with an associated error rate (see Section 5.2), the probability of certains events in the model can be determined by means of performability analysis.

6.1 Making the Model Reactive

An important aspect of probabilistic verification is that it requires the model to be *reactive*. This means that a component containing states must at some point wait for input or a change in the error model. Otherwise, the result of probabilistic verification may not be as expected: Transitions in the nominal model always have precedence over (probabilistic) transitions in the error model. To this end, we modify the discrete battery sensor model itself, such that state changes occur only if there is a change in the environment of the battery.

The changed components of the model are shown in Figure 6.1. The loops that update the status of the ports of the batteries have been replaced by a special type of connection called *flow*: These allow an expression composed of input ports and constant values to be connected to a single output port. For example, the connection in `Sensor.Imp` sets the value of `reading` to the negation of `has_power`, indicating a reading is available as long as there is power.

The inputs for the mode switching in the `System` component have been marked *blocking*. By default, SLIM will input-enable these ports, making it possible for the system to always accept these events even if the mode is already switched. By marking the ports as blocking, these inputs are accepted only in the `primary` mode.

6.2 Probabilistic Properties

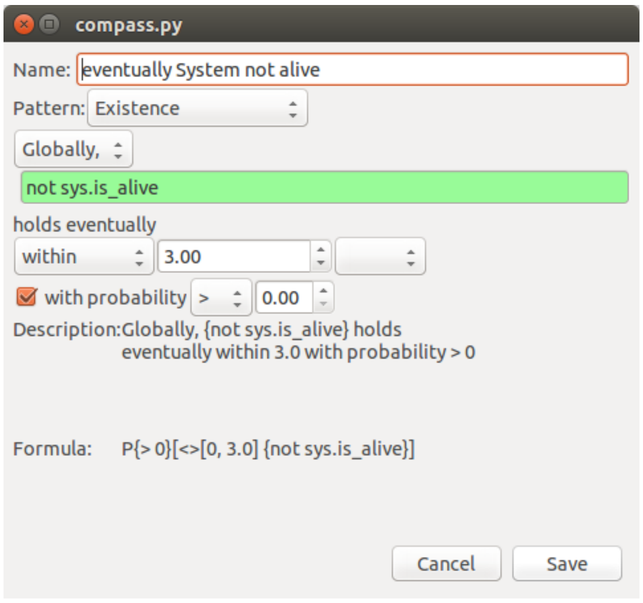
We will now add some property for performability analysis

- Click on the *Properties* tab
- Click on the item `Enclosure.Imp`
- Click on the *Properties* sub-tab
- Click *Add* in the *Patterns* sub-pane

```
system System
  features
    is_alive: out data port bool {Observable => true;};
    mode_selector: out data port enum(Primary, Secondary1, Secondary2)
        {Default => "Primary"; Observable => true;};
    go_to_secondary1: in event port {Blocking => true;};
    go_to_secondary2: in event port {Blocking => true;};
end System;

system implementation Battery.Imp
  connections
    flow true -> has_power_out in modes(full, low_single, low_double);
    flow false -> has_power_out in modes(empty);
    flow true -> low in modes(low_single, low_double);
    flow false -> low in modes(full, empty);
  states
    full: initial state;
    low_single: state;
    low_double: state;
    empty: state;
  transitions
    full -[ when not has_power_in and
         not double_consumption and
                 not zero_consumption ]-> low_single;
    full -[ when not has_power_in and
         double_consumption and
                 not zero_consumption ]-> low_double;
    low_single -[ when not zero_consumption ]-> empty;
    low_double -[ when not zero_consumption ]-> empty;
end Battery.Imp;

system implementation Sensor.Imp
  connections
    flow has_power -> reading;
  states
    base: initial state;
  properties
    ErrorModel => classifier(PermanentFailure.Imp);
    FaultEffects => ([State=>"dead"; Target=>reference(reading);
                       Effect=>"false";]);
end Sensor.Imp;
```

Figure 6.1: Battery Sensor model: basic components' implementation.

- Set the values as shown in Figure 6.2



Name: eventually System not alive
Pattern: Existence
Globally,
not sys.is_alive
holds eventually
within 3.00
✓ with probability > 0.00
Description: Globally, {not sys.is_alive} holds eventually within 3.0 with probability > 0
Formula: P{> 0}[<>[0, 3.0] {not sys.is_alive}]

Figure 6.2: A probabilistic property.

By having performed these steps, the `eventually not System is_alive` property is now a time-bounded probabilistic property, which may be used for performability analysis.

6.3 Performability Analysis

Actual performability analysis can be be performed using the *Analysis* sub-tab of the *Performability* tab. We will do so by taking the following steps

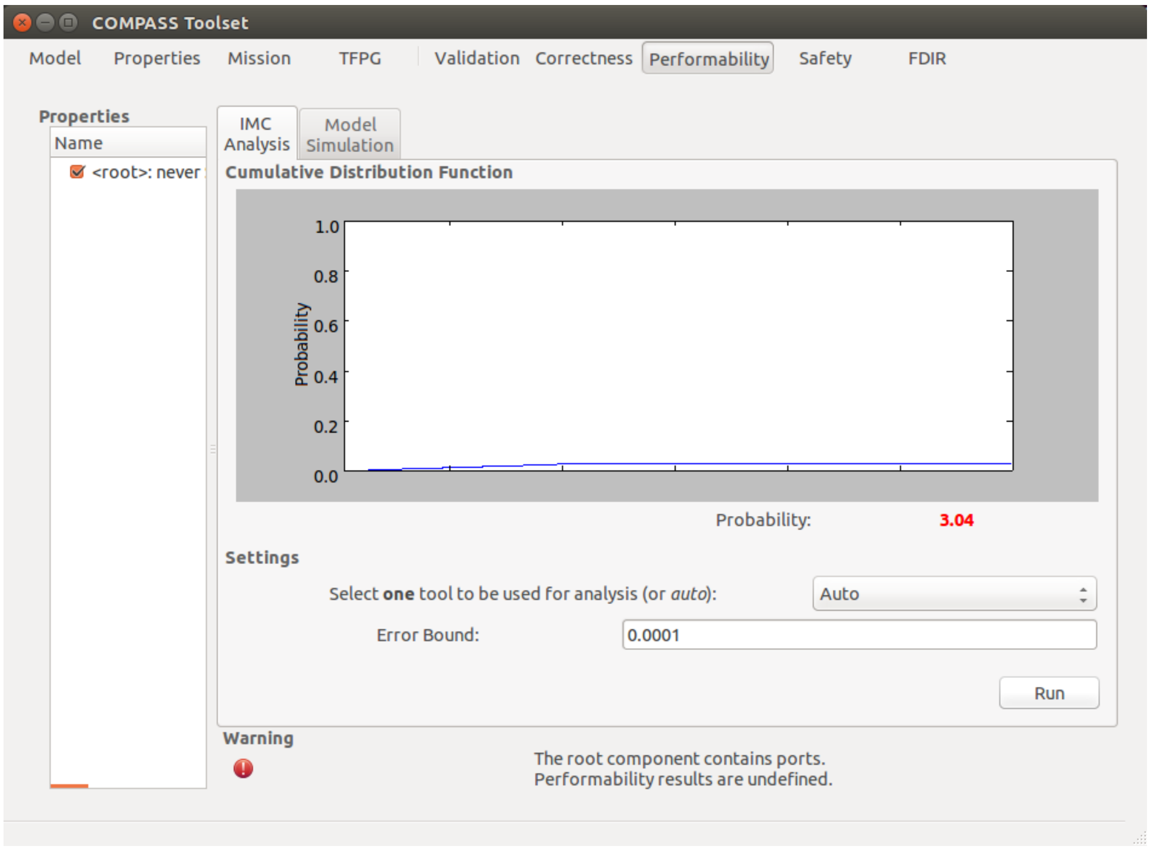
- Perform the steps of Section 6.2
- Click on the *Performability* tab
- Click on the *IMC Analysis* sub-tab
- In the left pane, check the `eventually not System is_alive` property
- On the right, set the *Error Bound* field to 0.0001
- Click *Run*

After some delay, a graph will be plotted that shows the (cumulative) probability distribution of the property holding true after a given delay, as shown in Figure 6.2.

6.4 Performability Simulation

Performability simulation can be performed using the *Simulation* sub-tab of the *Performability* tab. We will do so by taking the following steps

- Perform the steps of Section 6.2



Probability: 3.04

Select **one** tool to be used for analysis (or *auto*): Auto

Error Bound: 0.0001

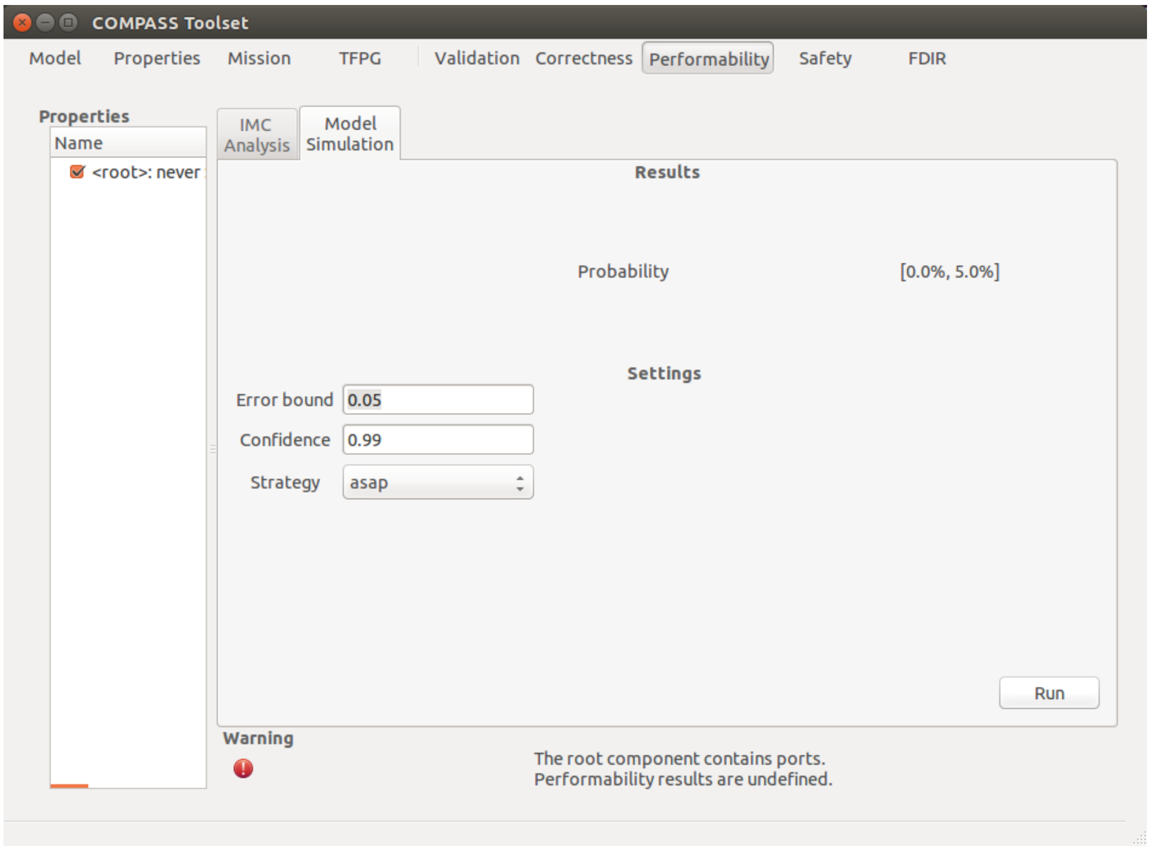
The root component contains ports.
Performability results are undefined.

Figure 6.3: Performing performability analysis.

- Click on the *Performability* tab
- Click on the *Model Simulation* sub-tab
- In the left pane, check the `eventually not System is_alive` property
- The simulation parameters can be left at their default values
- Click *Run*

After some delay, the overall probability estimated of the property holding true after within its specified time bound is given, as shown in Figure 6.4.

For more details about the differences between performability analysis and simulation, see Section 9.3



<root>: never

Results

Probability [0.0%, 5.0%]

Settings

Error bound 0.05

Confidence 0.99

Strategy asap

Warning

The root component contains ports.
Performability results are undefined.

Figure 6.4: Performing performability simulation.

Chapter 7

Fault Detection, Identification and Recovery

In this chapter, we discuss Fault Detection, Identification and Recovery (FDIR for short): we introduce the notions of observability and alarms, and we illustrate how to build an FDIR sub-system and verify its effectiveness in terms of detection, isolation and recovery capabilities. We also discuss diagnosability and the use of TFPGs for FDIR.

7.1 Modeling Observables

Observables indicate signals that are visible to an (FDIR) component. In SLIM, data ports may be tagged with the attribute *observable* for this purpose. Typically, observables can be used to model information that is provided by sensors.

We extend the battery sensor model with delays, by adding some observability information (see model `system_fdir.slim`). In particular, we tag the `low` output port of each battery as observable, with the idea of using this signal to detect a fault of the battery. Additionally, we also want to export these signals at system level, so that they can be accessible to the FDIR controller we want to model. We do this as shown in Figure 7.1. In order to model the routing to the system level, we add an output data port `low` to the PSU interface, and two output data ports `battery1_low` and `battery2_low` to the system interface, all of them tagged as observables. The connections are drawn in the corresponding component implementations.

7.2 Modeling a Controller

We can now use the observables defined in Section 7.1, in order to detect a fault of either of the two batteries. Eventually, when we detect such a fault, we want to re-configure the system, so that it is able to tolerate it. In particular, we can re-configure the system using the `secondary1` and `secondary2` system modes.

We model the controller (the FDIR component) as a component running in parallel, synchronously, with the system model, and connected with the system by means of the observables (the `battery1_low` and `battery2_low` signals) and by means of event ports (`go_to_secondary1` and `go_to_secondary2`) that can be triggered to issue the re-configuration. Notice, that as a difference with the models in Chapter 4, where these event ports were left

```
system Battery
  features
    [...]
    low: out data port bool {Default => "false"; Observable => true;};
end Battery;

system PSU
  features
    [...]
    low : out data port bool {Default => "false"; Observable => true;};
end PSU;

system System
  features
    [...]
    battery1_low: out data port bool {Default => "false";
                                      Observable => true;};
    battery2_low: out data port bool {Default => "false";
                                      Observable => true;};
end System;

system implementation PSU.Imp
  connections
    [...]
    port battery.low -> low;
end PSU.Imp;

system implementation System.Imp
  connections
    [...]
    port psu1.low -> battery1_low;
    port psu2.low -> battery2_low;
  [...]
end System.Imp;
```

Figure 7.1: Adding observability information and connections.

free, here we want to connect them to the FDIR controller.

We specify the controller as follows – see Figure 7.2. The controller is modeled by the `monitor` component. Its interface contains the observable input data ports, and the event ports to control the system. Notice the event ports are *output* ports (whereas they are *input* ports for the system), since they are triggered by the controller itself. The monitor implementation models a simple, time-triggered, controller that checks the status of the batteries each time unit. In case either battery has a low charge, it issues a proper re-configuration (e.g., `go_to_secondary2` is triggered when `battery1_low` is true, and moves from the `base` state to the `recovery` state). The AADL property `FDIR =>`**`true`** additionally tags the component

```
system Monitor
  features
    battery1_low : in data port bool {Default => "false";};
    battery2_low : in data port bool {Default => "false";};
    go_to_secondary1: out event port;
    go_to_secondary2: out event port;
  properties
    FDIR => true;
end Monitor;

system implementation Monitor.Imp
  subcomponents
    delay: data clock;
  states
    base: initial state while (delay <= 1);
    recovery: state while (delay <=1);
  transitions
    base -[ when delay >= 1 and not battery1_low and not battery2_low
            then delay := 0 ] -> base;
    base -[ go_to_secondary2 when delay >= 1 and battery1_low
                             then delay := 0 ]-> recovery;
    base -[ go_to_secondary1 when delay >= 1 and battery2_low
                             then delay := 0 ]-> recovery;
    recovery -[ when delay >= 1 then delay := 0 ]-> recovery;
  properties
    FDIR => true;
end Monitor.Imp;

system System
  features
   [...]
    go_to_secondary1: in event port;
    go_to_secondary2: in event port;
end System;

system Enclosure
end Enclosure;

system implementation Enclosure.Imp
  subcomponents
    sys: system System.Imp;
    mon: system Monitor.Imp;
  connections
    port sys.battery1_low -> mon.battery1_low;
    port sys.battery2_low -> mon.battery2_low;
    port mon.go_to_secondary1 -> sys.go_to_secondary1;
    port mon.go_to_secondary2 -> sys.go_to_secondary2;
end Enclosure.Imp;
```

Figure 7.2: Adding an FDIR controller.

as being an FDIR one.

In addition to modeling the controller, we modify, as it can also be seen from Figure 7.2, the `System` and the `Enclosure` components so that the `go_to_secondary1` and `go_to_secondary2` signals are declared as input ports of the system and routed from the controller to the system, instead of being connected to the external environment – this is visible in the implementation of the `Enclosure` component.

7.3 Safety Analysis Revisited

We show that, introducing the controller, has a beneficial effect on the safety assessment of the model. In particular, we run again Fault Tree Analysis (compare Section 5.8) on the new model.

Steps

- Load the model `system_fdir.slim` in the GUI
- Open the *File* menu and load the properties contained in file `examples/battery_sensor/system_fdir.propxml`
- Click on the *Safety* tab and select the *Fault Tree Generation* pane
- Only the property `TLE: Not is_alive` appears on the left; this is why it is the only propositional property
- Select the property
- Expand the *Model Checker Options* pane and check *Compute Probabilities*
- The bottom pane becomes editable; assign the following probabilities
  - `sys.psu1.generator._errorSubcomponent.__fault`: 0.2
  - `sys.psu2.generator._errorSubcomponent.__fault`: 0.2
  - `sys.sensor1._errorSubcomponent.__fault`: 0.3
  - `sys.sensor2._errorSubcomponent.__fault`: 0.3
- Click the *Generate Fault Tree* button
- A fault tree such as the one in Figure 7.3 is created;

As we can see from Figure 7.3, the single points of failure (failure of either of the two generators) have disappeared from the model. The rationale is that the re-configuration of the system has made the system more robust, in particular now it is resistant to a single fault of a generator. Probabilistically, the probability of the system not being alive is reduced from 0.418 (i.e, probability of the continuous model with no controller – it is identical to the probability for the discrete model) to 0.126.

Notice that the monitor checks the status of the batteries each time unit. If we increase the time bounds to 2 time units, the monitor will be unable to detect the battery failure in

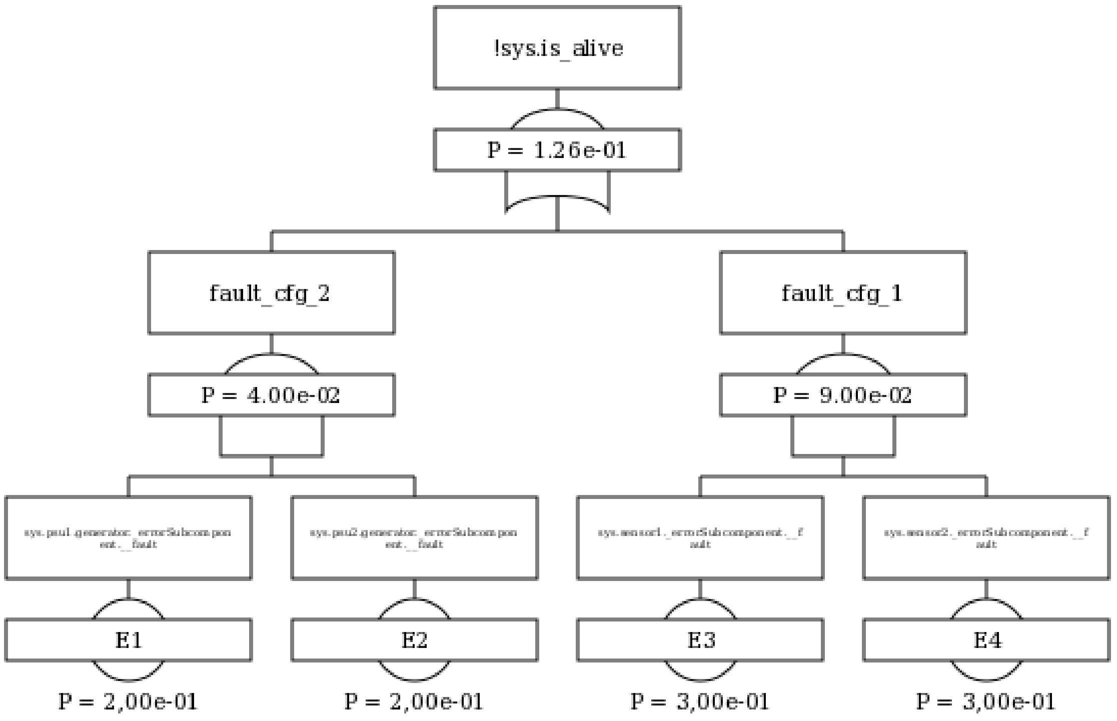

!sys.is_alive
P = 1.26e-01
fault_cfg_2
P = 4.00e-02
fault_cfg_1
P = 9.00e-02
sys.psu1.generator._errorSubcomponent.__fault
sys.psu2.generator._errorSubcomponent.__fault
sys.sensor1._errorSubcomponent.__fault
sys.sensor2._errorSubcomponent.__fault
E1
E2
E3
E4
P = 2,00e-01
P = 2,00e-01
P = 3,00e-01
P = 3,00e-01

Figure 7.3: Probabilistic Fault Tree Generation for `system_fdir` example.

time to make the re-configuration, since in the `low_single` state the battery will discharge completely in 2 time units. The user can verify that, in this situation, the fault tree contains additional minimal cut sets.

7.4 Modeling Alarms

An FDIR component may be associated with alarms. An alarm is typically associated with the detection of an anomaly, e.g. detection of a unexpected system behavior. Any Boolean outgoing data-port of an FDIR component may be tagged as being an alarm. Typically, an alarm may be designed so that it is triggered by a specific fault, or by a set or combinations of faults. When an alarm is triggered by a specific fault, we speak about *perfect fault isolation*.

As an example, we want add alarms to the monitor component, that are raised when a battery has a low charge. Each battery has its own associated alarm. We do this as shown in Figure 7.4.

In general, the purpose of alarms is to raise awareness of a system anomaly, so that appropriate countermeasures (e.g., re-configuration to recover from a fault) can be taken. In our simple battery sensor example, raising of the alarms and system re-configuration are carried out by the same component (`monitor`). More in general, these two actions may be carried out by separate (communicating) components, that is, a fault detection (FD) component is responsible to perform fault detection and raise the alarms, and a fault recovery (FR) component reads the alarms and reacts by triggering the appropriate recovery actions.

```
system Monitor
  features
    [...]
    alarm_battery1: out data port bool {Default => "false";
                                        Alarm => true;};
    alarm_battery2: out data port bool {Default => "false";
                                        Alarm => true;};
  [...]
end Monitor;

system implementation Monitor.Imp
  [...]
  transitions
    base -[ when delay >= 1 and not battery1_low and not battery2_low
            then delay := 0 ] -> base;
    base -[ go_to_secondary2 when delay >= 1 and battery1_low
            then delay := 0; alarm_battery1 := true ]-> recovery;
    base -[ go_to_secondary1 when delay >= 1 and battery2_low
            then delay := 0; alarm_battery2 := true ]-> recovery;
    recovery -[ when delay >= 1 then delay := 0 ]-> recovery;
  [...]
end Monitor.Imp;
```

Figure 7.4: Adding alarms.

7.5 FDIR Effectiveness Analysis

The purpose of FDIR effectiveness analysis is to assess the effectiveness of an existing FDIR component. For instance, we may assess its capability to detect a specific fault or anomaly (effectiveness of detection/identification), or its effectiveness to recover from it (effectiveness of recovery).

One way to assess such effectiveness is to model check the system with respect to a suitable set of properties.

For instance, let us check the following property: `B1_LOW ->ALARM_B1` (charge level low of `battery1` eventually followed by alarm of `battery1`). This property formalizes the notion of completeness of the detection (condition implies that eventually the alarm will be raised).

Steps

- Load the model `system_fdir.slim` in the GUI
- Open the *File* menu and load the properties contained in file `examples/battery_sensor/system_fdir.propxml`
- Click on the *Correctness* tab and select the *Model Checking* pane
- The following properties are shown on the left *Properties* pane:

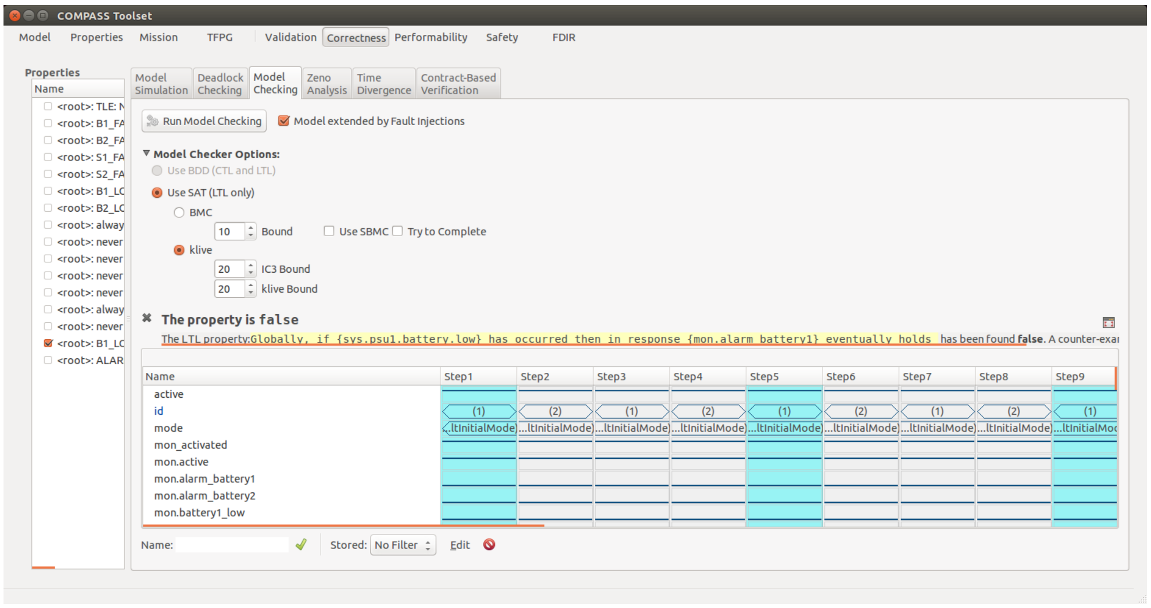
  - `TLE: Not is_alive`
  - `B1_LOW`
  - `B2_LOW`
  - `B1_or_B2_LOW`
  - `always System is_alive`
  - `never PSUA Low`
  - `never PSUA empty`
  - `never Not is_alive`
  - `never Secondary1`
  - `always Generator.has_power`
  - `never Low double`
  - `B1_LOW -> ALARM_B1`
  - `always ALARM_B1 implies B1_LOW`
  - `ALARM_B1 -> SECONDARY2`
  - `Battery low responded by reconf`

- Select the property `B1_LOW -> ALARM_B1`.
- Enable the *Model Extended by Fault Injections* checkbox; other properties appear in the left hand side list
- Expand the element *Model Checker Options*; *klive* is selected as engine
- Set the values for *IC3 Bound* and *klive Bound* to 20
- Click the *Run Model Checking* button
- The property results to be false (see Figure 7.5)

It may seem unexpected, but the property is false. In fact, `battery1` may happen to discharge after the system has already been re-configured to mode `secondary1` (due to low charge of `battery2`), hence in this case the alarm for `battery2` will be raised, instead of that for `battery1`. We leave it as an exercise for the reader to try and weaken this property, in order to obtain one that is verified by the system (e.g., trying to detect the disjunction of two faults, or assuming the hypothesis of single fault).

We remark that the choice of a proper (sufficiently high) SAT bound may be important – e.g., using a SAT bound equal to 10 is not enough to falsify the property in the former example, as the user can check. The bound depends on the number of transitions in the model that are necessary to reach the desired counterexample. See also Chapter 9 for hints on dealing with the SAT bound.

Property `always ALARM_B1` **`implies`** `B1_LOW`, is instead true. This is a property stating the correctness of detection (raising of the alarm implies condition). Here, we are exploiting the fact that the signal `B1_LOW` is never reset – in a more general case, one should state that raising of the alarm implies that the condition was true *sometime in the past*. The property can be verified as follows.



[✓ Model extended by Fault Injections]

▼ Model Checker Options:

Use BDD (CTL and LTL)

[✓ Use SAT (LTL only)]

BMC

10 Bound ☐ Use SBMC ☐ Try to Complete

[✓ klive]

20 IC3 Bound

20 klive Bound

✖ **The property is false**

The LTL property:`Globally, if {sys.psu1.battery.low} has occurred then in response {mon.alarm_battery1} eventually holds` has been found **false**. A counter-exa

Name	Step1	Step2	Step3	Step4	Step5	Step6	Step7	Step8	Step9
active									
id	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)	(1)
mode	...ltInitialMode)	...ltInitialMode)	...ltInitialMode)	...ltInitialMode)	...ltInitialMode)	...ltInitialMode)	...ltInitialMode)	...ltInitialMode)	...ltInitialMoc
mon_activated									
mon.active									
mon.alarm_battery1									
mon.alarm_battery2									
mon.battery1_low									

Figure 7.5: Counterexample trace created for the property `B1_LOW -> ALARM_B1` of the `system_fdir` example (extended version).

Steps

- Load the model `system_fdir.slim` in the GUI
- Open the *File* menu and load the properties contained in file `examples/battery_sensor/system_fdir.propxml`
- Click on the *Correctness* tab and select the *Model Checking* pane
- Select the property `always ALARM_B1 implies B1_LOW`.
- Enable the *Model Extended by Fault Injections* checkbox; other properties appear in the left hand side list
- Expand the element *Model Checker Options*; *klive* is selected as engine
- Set the values for *IC3 Bound* and *klive Bound* to 20
- Click the *Run Model Checking* button
- The property results to be true

Note that in this case, the answer of the model checker is conclusive ("The property is true" and not "The property is true up to bound ...").

Property `ALARM_B1 ->SECONDARY2` (raising of the alarm for `battery1` is eventually followed by a re-configuration to mode `secondary2`) is also true.

COMPASS provides additional, specific tasks to carry out FDIR effectiveness analysis, called *Fault Detection Analysis*, *Fault Isolation Analysis* and *Fault Recovery Analysis*. They are illustrated below.

7.5.1 Fault Detection Analysis

Fault detection analysis has the purpose to analyze whether a given condition (e..g, a fault or anomaly) can indeed be detected using any of the system alarms. Fault detection analysis takes as input the condition, and returns as output the *detection means*, that is, the set of alarms that are necessarily (and eventually) raised when the condition is true.

For instance, we could analyze fault detection for the two properties B1_LOW and B2_LOW that specify the fact that the charge level of the two batteries is low. However, we already know (from the results we obtained using model checking at the beginning of Section 7.5) that it is not possible to detect such conditions. Then, we can try and add a further alarm to the model, namely one that detects the failure of either battery. We do it as in Figure 7.6.

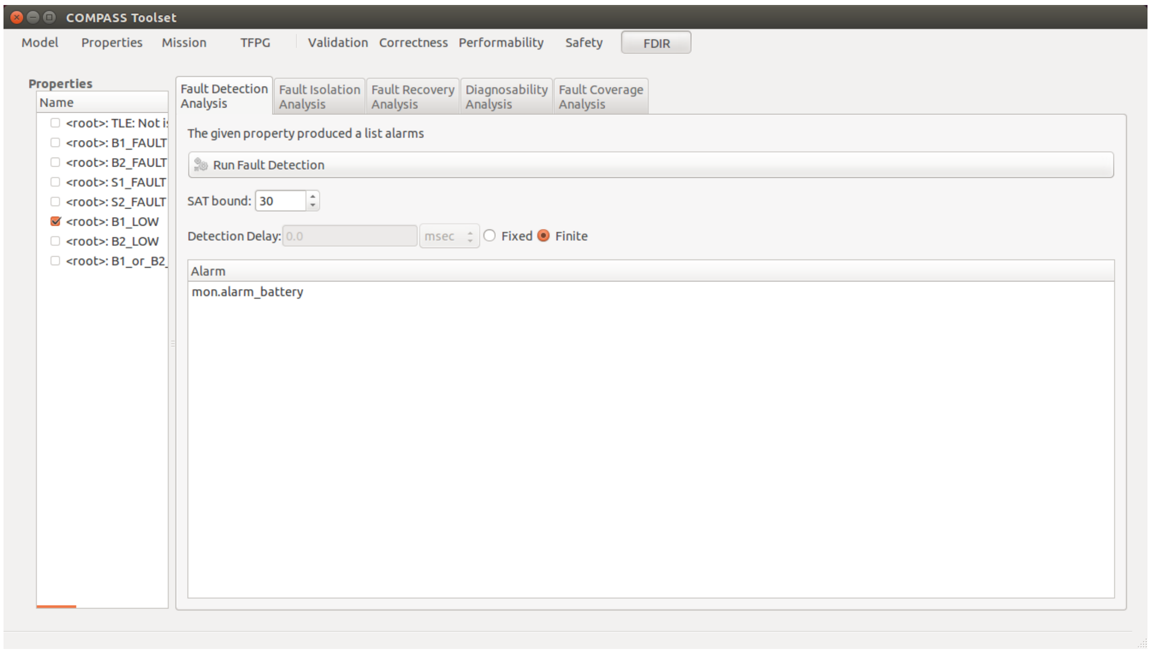
```
system Monitor
  features
    [...]
    alarm_battery: out data port bool {Default => "false";
                                       Alarm => true;};
  [...]
end Monitor;

system implementation Monitor.Imp
  [...]
  connections
    flow true -> alarm_battery in modes (recovery);
  [...]
end Monitor.Imp;
```

Figure 7.6: Adding alarms.

We now run fault detection for the property B1_LOW. We do it as follows.

Steps

- Load the model system_fdir.slim in the GUI
- Open the *File* menu and load the properties contained in file examples/battery_sensor/system_fdir.propxml
- Click on the *FDIR* tab and select the *Fault Detection Analysis* pane
- Select the property B1_LOW on the left
- Set the value for the *SAT bound* to 30
- Click the *Run Fault Detection* button
- The *Alarm* pane is filled with the following list: mon.alarm_battery (see Figure 7.7)



<root>: TLE: Not i
<root>: B1_FAULT
<root>: B2_FAULT
<root>: S1_FAULT
<root>: S2_FAULT
<root>: B1_LOW ✓
<root>: B2_LOW
<root>: B1_or_B2

The given property produced a list alarms

SAT bound: 30

Detection Delay: 0.0 msec ○ Fixed ● Finite

Alarm
mon.alarm_battery

Figure 7.7: Fault Detection Analysis using property `B1_LOW` of the `system_fdir` example.

Again, the choice of a proper (sufficiently high) SAT bound is important – e.g., using a lower SAT bound may produce inaccurate results (spurious alarms may be returned as possible detection means). In this case, the bound depends on the number of transitions in the model that are necessary to rule out the wrong detection means and on the underlying model checking algorithm.

7.5.2 Fault Isolation Analysis

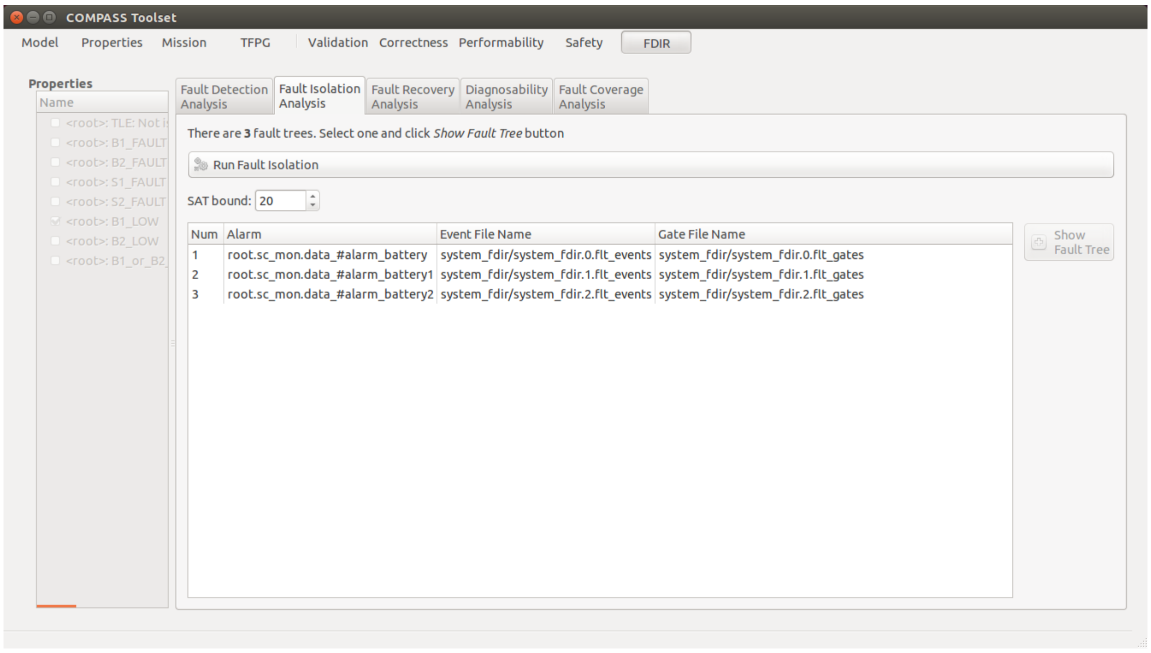
Fault isolation analysis has the purpose to analyze the conditions under which the alarms may be triggered. Namely, it takes as input the set of the alarms defined in the model and, for each alarm, it generates a fault tree, showing the minimal combinations of faults that may trigger it. We speak about *perfect isolation* in case the fault tree contains only one minimal cut set of cardinality one – in other words, the alarm may be triggered by one specific fault.

For instance, we can run fault isolation as follows.

Steps

- Load the model `system_fdir.slim` in the GUI
- Click on the *FDIR* tab and select the *Fault Isolation Analysis* pane
- Set the value of *SAT Bound* to 20
- Click the *Run Fault Isolation* button; the analysis produces three fault trees (see Figure 7.8)
- Select the first row, the one having `root.sc_mon.data_#alarm_battery` in column *Alarm*
- Click the *Show Fault Tree* button; the fault tree in Figure 7.9 appears.

As the reader can see, we have perfect isolation for the alarms `alarm_battery1` and `alarm_battery2`, but not for the alarm `alarm_battery`.



There are **3** fault trees. Select one and click *Show Fault Tree* button

SAT bound: 20

Num	Alarm	Event File Name	Gate File Name
1	root.sc_mon.data_#alarm_battery	system_fdir/system_fdir.0.flt_events	system_fdir/system_fdir.0.flt_gates
2	root.sc_mon.data_#alarm_battery1	system_fdir/system_fdir.1.flt_events	system_fdir/system_fdir.1.flt_gates
3	root.sc_mon.data_#alarm_battery2	system_fdir/system_fdir.2.flt_events	system_fdir/system_fdir.2.flt_gates

Figure 7.8: Fault Isolation Analysis for example `system_fdir`.

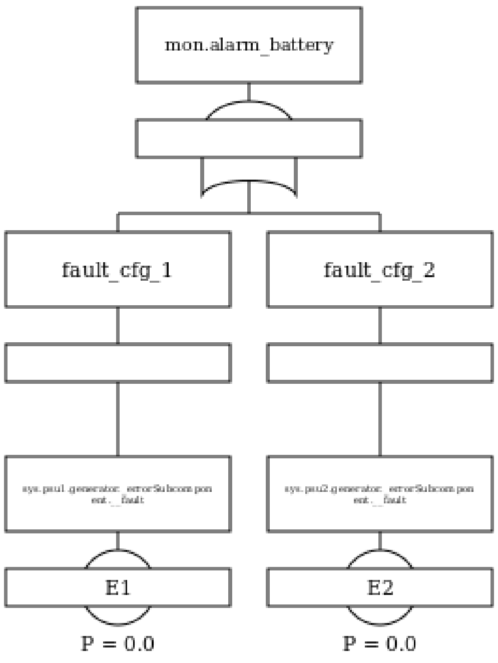

mon.alarm_battery
fault_cfg_1
fault_cfg_2
sys.psu1.generator._errorSubcomponent._fault
sys.psu2.generator._errorSubcomponent._fault
E1
E2
P = 0.0
P = 0.0

Figure 7.9: Fault Tree generated by Fault Isolation Analysis for the `alarm_battery` alarm.

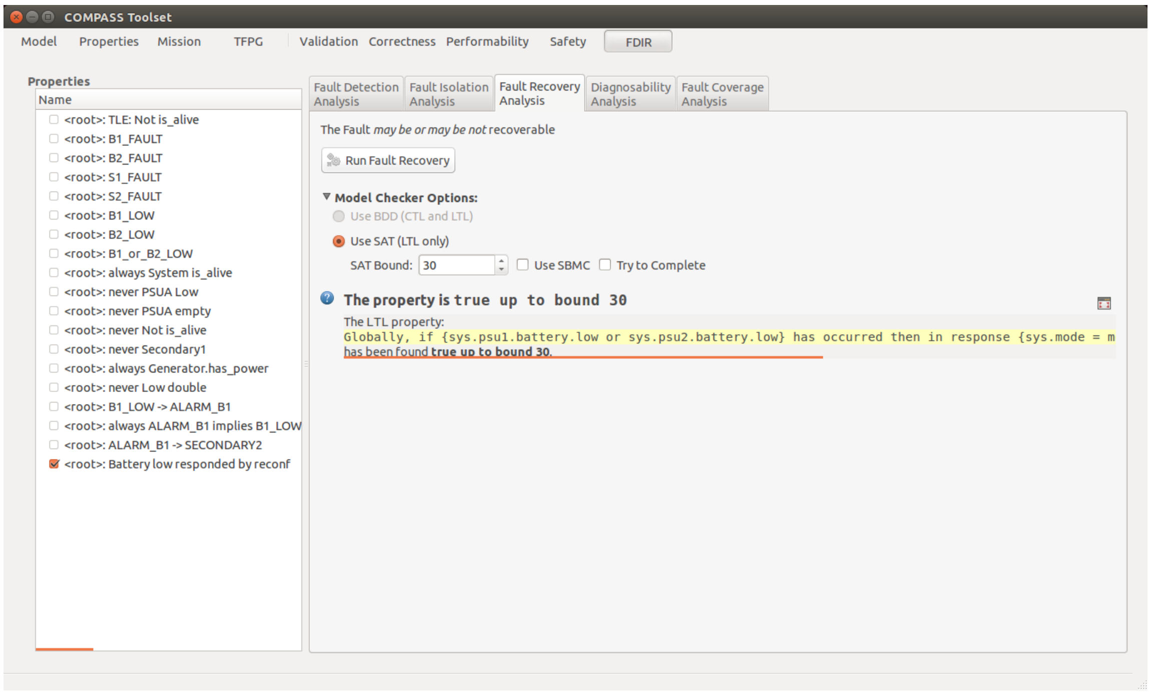
7.5.3 Fault Recovery Analysis

Fault recovery analysis is similar in spirit to model checking. It is possible to write a property representing recoverability of the system, and to check it against the model.

For instance, let us verify the following property: `Battery low responded by reconf`. It states that whenever either battery is in state low, it will trigger a re-configuration of the system into one of the secondary modes.

Steps

- Load the model `system_fdir.slim` in the GUI
- Open the *File* menu and load the properties contained in file `examples/battery_sensor/system_fdir.propxml`
- Click on the *FDIR* tab and select the *Fault Recovery Analysis* pane
- On the *Properties* pane, select the property `Battery low responded by reconf`
- Expand the element *Model Checker Options*, select *SAT* engine and set the *SAT Bound* to 30
- Click the *Run Fault Recovery* button; the property is true up to the selected bound (see Figure 7.10).



Properties

Name
☐ <root>: TLE: Not is_alive
☐ <root>: B1_FAULT
☐ <root>: B2_FAULT
☐ <root>: S1_FAULT
☐ <root>: S2_FAULT
☐ <root>: B1_LOW
☐ <root>: B2_LOW
☐ <root>: B1_or_B2_LOW
☐ <root>: always System is_alive
☐ <root>: never PSUA Low
☐ <root>: never PSUA empty
☐ <root>: never Not is_alive
☐ <root>: never Secondary1
☐ <root>: always Generator.has_power
☐ <root>: never Low double
☐ <root>: B1_LOW -> ALARM_B1
☐ <root>: always ALARM_B1 implies B1_LOW
☐ <root>: ALARM_B1 -> SECONDARY2
✓ <root>: Battery low responded by reconf

The Fault *may be or may not* recoverable

Model Checker Options:

Use BDD (CTL and LTL) [✓ Use SAT (LTL only)]

SAT Bound: 30 ☐ Use SBMC ☐ Try to Complete

The property is `true up to bound 30`

The LTL property:

```
Globally, if {sys.psu1.battery.low or sys.psu2.battery.low} has occurred then in response {sys.mode = m
```
has been found **true up to bound 30.**

Figure 7.10: Fault Recovery Analysis for property `Battery low responded by reconf`.

7.6 Diagnosability Analysis

Diagnosability analysis is a fundamental step to investigate the degree of observability of a given system. Given a condition to be diagnosed (called *diagnosability condition*), diagnosability analysis asks whether the system has enough observables to be able to carry out the diagnosis. In other words, diagnosability analysis checks whether there exists a diagnoser that is capable of carrying out the diagnosis, using the available observations.

The diagnosability problem is different from the one of building the diagnoser itself. In this tutorial, for convenience, we illustrate diagnosability analysis at this point, in such a way that the user is already familiar with the problem of building a diagnoser and with the properties that are required from it. However, notice that typically, diagnosability analysis is performed earlier in the design process, before designing the actual diagnoser – the system being diagnosable may be seen as a precondition to build such a diagnoser.

The level of observability can be thought as the set or sensors that are available in the system – the idea is that, having more sensors implies potentially increased diagnosis capabilities, but also may increase the cost of building the system. Diagnosability analysis may help the designer to address the problem of identifying the sensors that are needed to implement FDIR for a specific systems. Also, it may help the designer to investigate the trade-off between diagnoser effectiveness and costs.

As an example, we check diagnosability analysis for the condition `B1_LOW` (`battery1` level is low). We do it as follows.

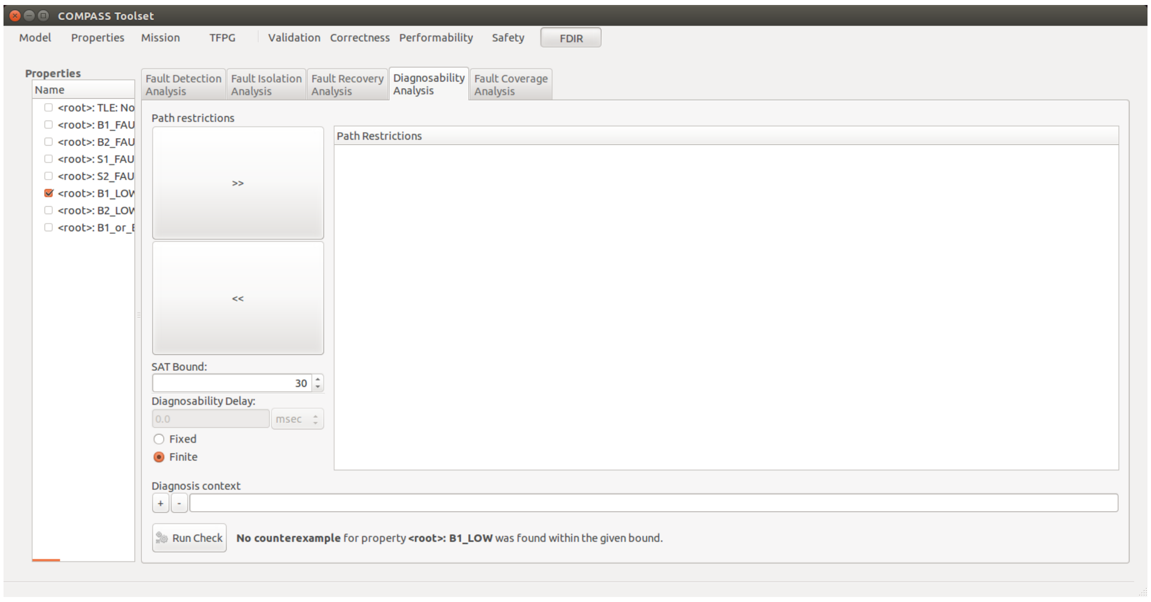
Steps

- Load the model `system_fdir.slim` in the GUI
- Open the *File* menu and load the properties contained in file `examples/battery_sensor/system_fdir.propxml`
- Click on the *FDIR* tab and select the *Diagnosability Analysis* pane
- On the *Properties* pane, select the property `B1_LOW`
- Set the *SAT Bound* to 30
- Click the *Run Check* button; No counterexample is found – indeed this condition is diagnosable (see Figure 7.11).

In a similar way, the user can verify that also the condition `B1_FAULT` is diagnosable.

If we try with, instead, the condition `S1_FAULT` (`sensor1` is faulty), we get a counterexample. (see Figure 7.12) Indeed, we have no observables in the model that we can use to detect a sensor fault. (Notice that the `sys.is_alive` signal being observable does not help, since a failure of a single sensor will not trigger it).

A counterexample to diagnosability is shown as a pair of traces – see Figure 7.13. The pair of traces represents an execution of the so-called *twin plant*, that is, two copies of the system running in parallel, synchronously, that are observationally indistinguishable. In other words, the two traces represent two different executions of the system that cannot be distinguished (they have the same values for the observables) but such that the diagnosability condition is true in one of them but not in the other. In this example, in the first trace `sensor1` has failed



Properties

Name
☐ <root>: TLE: No
☐ <root>: B1_FAU
☐ <root>: B2_FAU
☐ <root>: S1_FAU
☐ <root>: S2_FAU
☑ <root>: B1_LOW
☐ <root>: B2_LOW
☐ <root>: B1_or_I

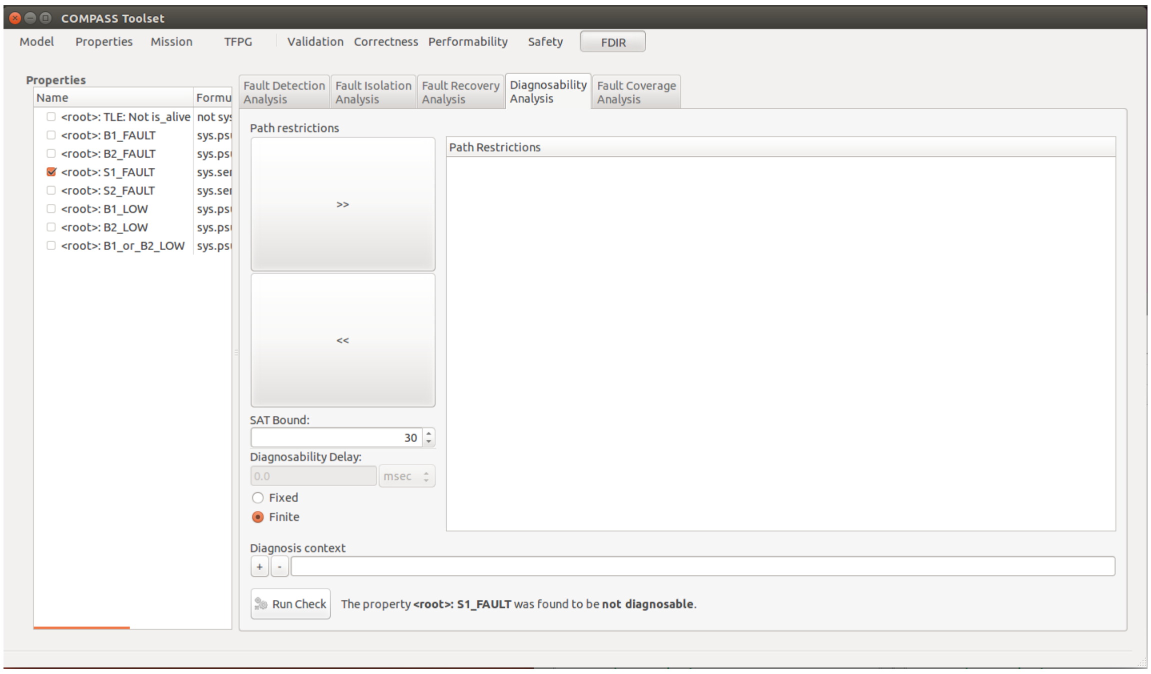
SAT Bound: 30

Diagnosability Delay: 0.0 msec

○ Fixed
● Finite

No counterexample for property **<root>: B1_LOW** was found within the given bound.

Figure 7.11: Diagnosability Analysis for property B1_LOW.



Properties

Name	Formu
☐ <root>: TLE: Not is_alive	not sys
☐ <root>: B1_FAULT	sys.psı
☐ <root>: B2_FAULT	sys.psı
☑ <root>: S1_FAULT	sys.sei
☐ <root>: S2_FAULT	sys.sei
☐ <root>: B1_LOW	sys.psı
☐ <root>: B2_LOW	sys.psı
☐ <root>: B1_or_B2_LOW	sys.psı

SAT Bound: 30

Diagnosability Delay: 0.0 msec

○ Fixed
● Finite

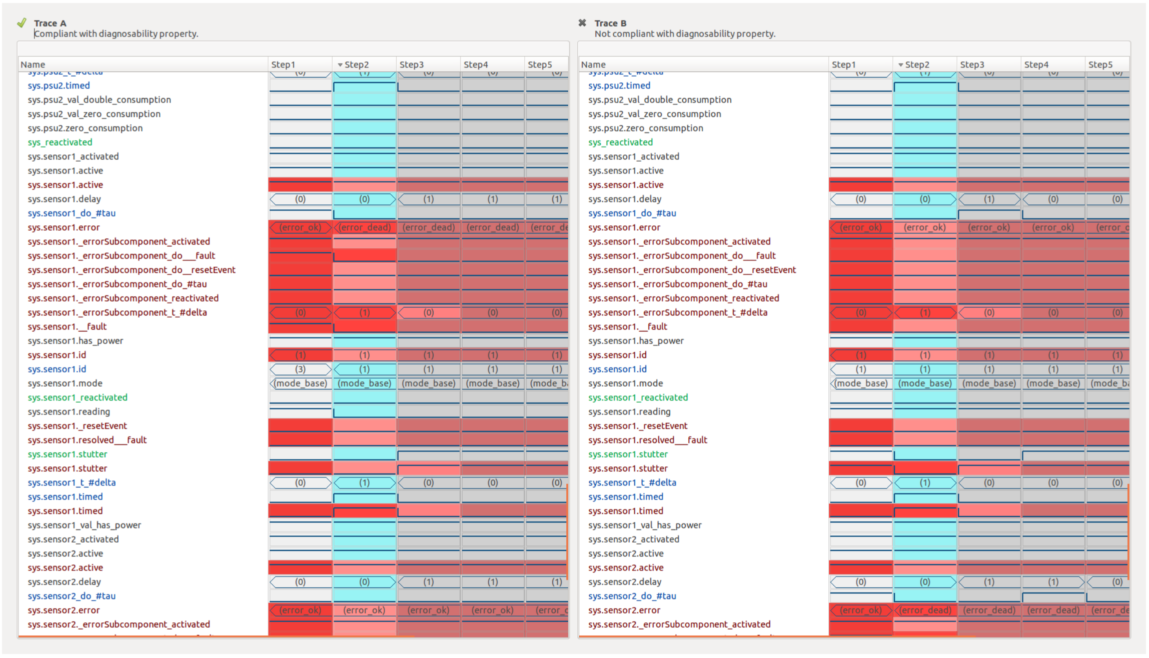
The property **<root>: S1_FAULT** was found to be **not diagnosable**.

Figure 7.12: Diagnosability Analysis for property S1_FAULT.

whereas in the second trace it is not failed, however there is no observable that can be used to distinguish the two situations.

7.7 Using TFPGs for Diagnosis

Timed Failure Propagation Graphs (compare Section 5.12) can be used as a model for diagnosis. The monitored discrepancies, that we mentioned in Section 5.12, represent conditions



Trace A
Compliant with diagnosability property.

Name	Step1	Step2	Step3	Step4	Step5
sys.psu2.timed					
sys.psu2_val_double_consumption					
sys.psu2_val_zero_consumption					
sys.psu2.zero_consumption					
sys_reactivated					
sys.sensor1_activated					
sys.sensor1.active					
sys.sensor1.active					
sys.sensor1.delay	(0)	(0)	(1)	(1)	(1)
sys.sensor1_do_#tau					
sys.sensor1.error	(error_ok)	(error_dead)	(error_dead)	(error_dead)	(error_de
sys.sensor1._errorSubcomponent_activated					
sys.sensor1._errorSubcomponent_do___fault					
sys.sensor1._errorSubcomponent_do__resetEvent					
sys.sensor1._errorSubcomponent_do_#tau					
sys.sensor1._errorSubcomponent_reactivated					
sys.sensor1._errorSubcomponent_t_#delta	(0)	(1)	(0)	(0)	(0)
sys.sensor1.__fault					
sys.sensor1.has_power					
sys.sensor1.id	(1)	(1)	(1)	(1)	(1)
sys.sensor1.id	(3)	(1)	(1)	(1)	(1)
sys.sensor1.mode	(mode_base)	(mode_base)	(mode_base)	(mode_base)	(mode_ba
sys.sensor1_reactivated					
sys.sensor1.reading					
sys.sensor1._resetEvent					
sys.sensor1.resolved___fault					
sys.sensor1.stutter					
sys.sensor1.stutter					
sys.sensor1_t_#delta	(0)	(1)	(0)	(0)	(0)
sys.sensor1.timed					
sys.sensor1.timed					
sys.sensor1_val_has_power					
sys.sensor2_activated					
sys.sensor2.active					
sys.sensor2.active					
sys.sensor2.delay	(0)	(0)	(1)	(1)	(1)
sys.sensor2_do_#tau					
sys.sensor2.error	(error_ok)	(error_ok)	(error_ok)	(error_ok)	(error_
sys.sensor2._errorSubcomponent_activated					

Trace B
Not compliant with diagnosability property.

Name	Step1	Step2	Step3	Step4	Step5
sys.psu2.timed					
sys.psu2_val_double_consumption					
sys.psu2_val_zero_consumption					
sys.psu2.zero_consumption					
sys_reactivated					
sys.sensor1_activated					
sys.sensor1.active					
sys.sensor1.active					
sys.sensor1.delay	(0)	(0)	(1)	(0)	(0)
sys.sensor1_do_#tau					
sys.sensor1.error	(error_ok)	(error_ok)	(error_ok)	(error_ok)	(error_
sys.sensor1._errorSubcomponent_activated					
sys.sensor1._errorSubcomponent_do___fault					
sys.sensor1._errorSubcomponent_do__resetEvent					
sys.sensor1._errorSubcomponent_do_#tau					
sys.sensor1._errorSubcomponent_reactivated					
sys.sensor1._errorSubcomponent_t_#delta	(0)	(1)	(0)	(0)	(0)
sys.sensor1.__fault					
sys.sensor1.has_power					
sys.sensor1.id	(1)	(1)	(1)	(1)	(1)
sys.sensor1.id	(1)	(1)	(1)	(1)	(1)
sys.sensor1.mode	(mode_base)	(mode_base)	(mode_base)	(mode_base)	(mode_ba
sys.sensor1_reactivated					
sys.sensor1.reading					
sys.sensor1._resetEvent					
sys.sensor1.resolved___fault					
sys.sensor1.stutter					
sys.sensor1.stutter					
sys.sensor1_t_#delta	(0)	(1)	(0)	(0)	(0)
sys.sensor1.timed					
sys.sensor1.timed					
sys.sensor1_val_has_power					
sys.sensor2_activated					
sys.sensor2.active					
sys.sensor2.active					
sys.sensor2.delay	(0)	(0)	(1)	(1)	(0)
sys.sensor2_do_#tau					
sys.sensor2.error	(error_ok)	(error_dead)	(error_dead)	(error_dead)	(error_de
sys.sensor2._errorSubcomponent_activated					

Figure 7.13: Diagnosability counterexample for property `S1_FAULT`.

that are observable, whereas non-monitored discrepancies represent conditions that are not observable.

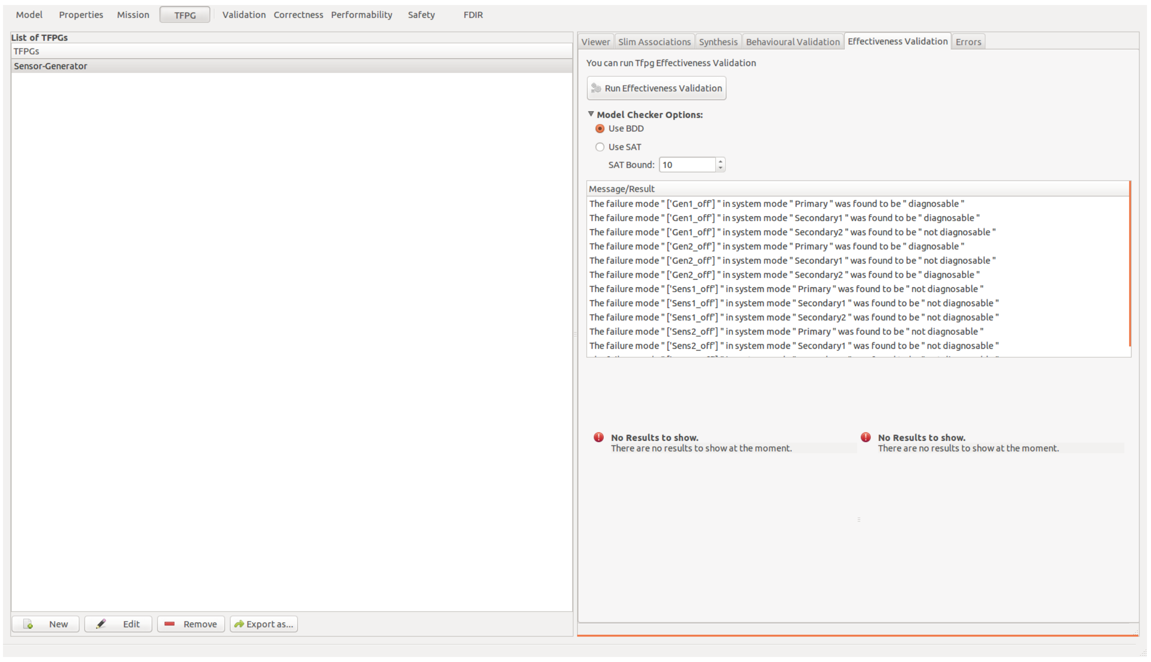
A TFPG model can be analyzed in terms of diagnosability effectiveness – that is, as a model for diagnosability. The failure modes and the propagation paths in the TFPG are analyzed, in order to check whether the given monitoring information is sufficient to detect the failure modes.

We can run diagnosability effectiveness analysis as follows.

Steps

- Load the model `system_fdir.slim` in the GUI
- Open the *File* menu and load the TFPG contained in file `examples/battery_sensor/system.txml`
- Click on the *TFPG* tab and select the *Effectiveness Validation* pane; expanding the element *Model Checker Options*, *Use BDD* is selected
- Click the *Run Effectiveness Validation* button; the pane is filled with the results as shown in Figure 7.14

The result can be interpreted as follows. *Gen1_off* is diagnosable (using the *B1_LOW* discrepancy) in modes Primary and Secondary1 but not in mode Secondary2. Indeed, in mode Secondary2, the first battery is not discharging, since it is disconnected from the sensor. Similarly for *Gen2_off*. On the other hand, *Sens1_off* and *Sens2_off* are not diagnosable in any mode, since there is no monitoring information that can be used to detect such faults.



You can run Tfpg Effectiveness Validation

Model Checker Options:
[✓ Use BDD] Use SAT
SAT Bound: 10

```
Message/Result
The failure mode " ['Gen1_off'] " in system mode " Primary " was found to be " diagnosable "
The failure mode " ['Gen1_off'] " in system mode " Secondary1 " was found to be " diagnosable "
The failure mode " ['Gen1_off'] " in system mode " Secondary2 " was found to be " not diagnosable "
The failure mode " ['Gen2_off'] " in system mode " Primary " was found to be " diagnosable "
The failure mode " ['Gen2_off'] " in system mode " Secondary1 " was found to be " not diagnosable "
The failure mode " ['Gen2_off'] " in system mode " Secondary2 " was found to be " diagnosable "
The failure mode " ['Sens1_off'] " in system mode " Primary " was found to be " not diagnosable "
The failure mode " ['Sens1_off'] " in system mode " Secondary1 " was found to be " not diagnosable "
The failure mode " ['Sens1_off'] " in system mode " Secondary2 " was found to be " not diagnosable "
The failure mode " ['Sens2_off'] " in system mode " Primary " was found to be " not diagnosable "
The failure mode " ['Sens2_off'] " in system mode " Secondary1 " was found to be " not diagnosable "
```

No Results to show.
There are no results to show at the moment.

No Results to show.
There are no results to show at the moment.

Figure 7.14: Effectiveness Validation for example `system_fdir`.

Chapter 8

Contract-Based Design

A SLIM model is a component-based description of the system: component types define the interfaces, component implementations define the internal structure and recursive decomposition in terms of other components, and the system is the root of such decomposition tree. Contract-based design allows to enrich this component-based specification with contracts, which specify the properties that each component assumes satisfied by the environment where it is instantiated and the properties must be guaranteed in response by any component implementation.

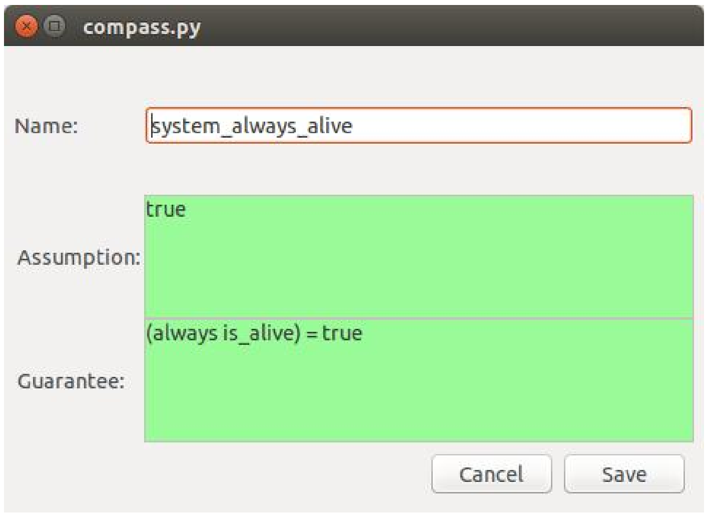
In this chapter, we explain how to specify contracts with SLIM and how to use COMPASS to perform contract-based analysis such as checking the contract refinement and generating hierarchical fault trees from the contract specification.

8.1 Contracts Specification

The first step is to specify contracts in the SLIM model. Contracts can be specified using the GUI or directly in the SLIM model. Let us define a contract for the system component of the `system_discrete` example using the GUI. As before, we want to specify that the system is always alive without assumptions on the environment, that is, the atomic proposition `sys.is_alive` is always true. We do so as follows.

Steps

- Load the model as shown in Section 4.4.1
- Click on the *Properties* tab; the *Requirements* pane is shown
- On the left, select the component `System`
- Click on the *Contracts* sub-pane
- Click the bottom-most *Add* button; specify a name for the contract; leave the assumption `true`; replace the guarantee `true` with the property “`always is_alive=true` as shown in figure 8.1
- Click the *Save* button; the name of the properties appears in the bottom-most pane, and a “(1)” is shown next to the name `System` on the left to specify that one property has been added for that component (see Figure 8.2)

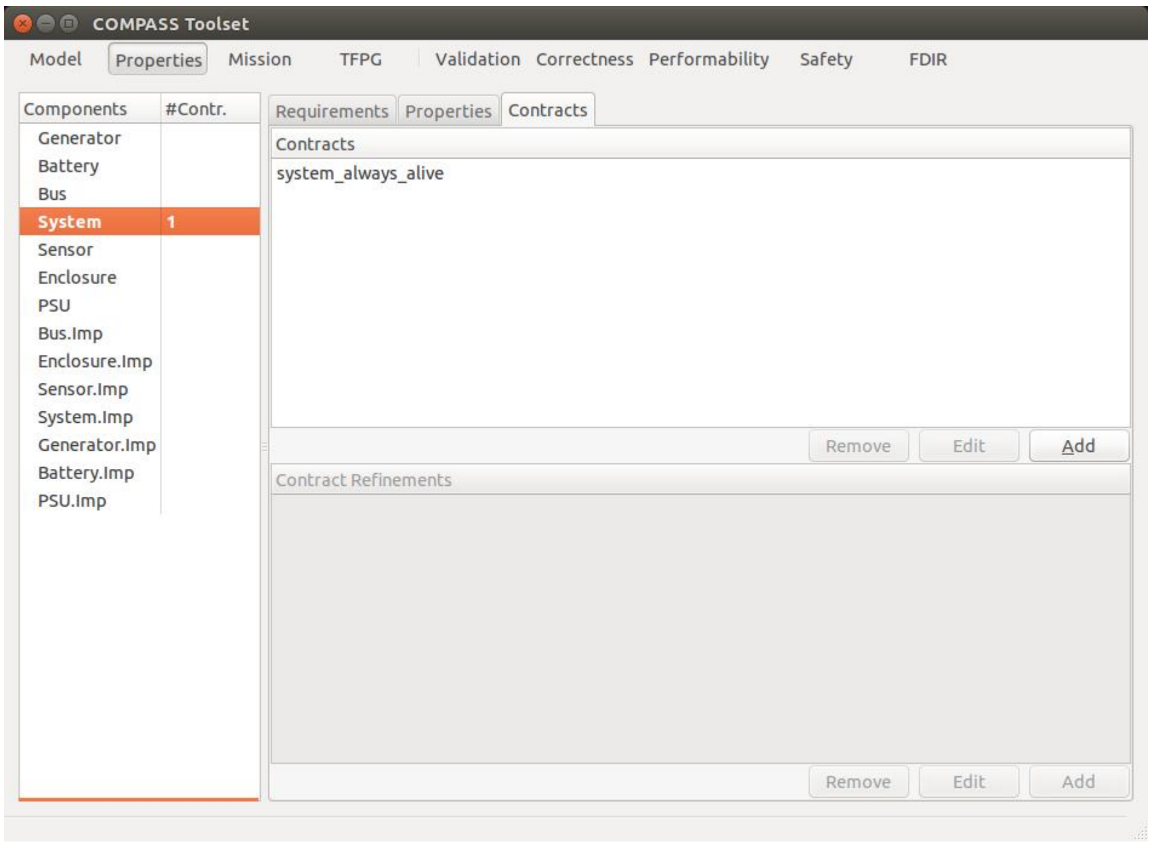


Name: system_always_alive

Assumption: true

Guarantee: (always is_alive) = true

Figure 8.1: Adding the contract `system_always_alive` for the `system_discrete` example.



Components	#Contr.
Generator	
Battery	
Bus	
System	1
Sensor	
Enclosure	
PSU	
Bus.Imp	
Enclosure.Imp	
Sensor.Imp	
System.Imp	
Generator.Imp	
Battery.Imp	
PSU.Imp	

Contracts
system_always_alive

Figure 8.2: *Contracts* pane after the contract `system_always_alive` has been added.

You can also specify the contracts directly in the SLIM model. Let us now consider the `system_contracts` example, which already contains a full specification of contracts for the different components. So, for example Figure 8.3 shows the battery sensor model with two contracts specified on the `System` component.

```
...
system System
  features
    is_alive: out data port bool;
  properties
    SLIMpropset::Contracts => ([Name => "alive";
    Assumption => "true";
    Guarantee => "always(is_alive)";
    ],
    [Name => "delay";
    Assumption => "true";
    Guarantee => "(always (is_alive)) or
                  (time_until(fall(is_alive))>=4))";]);
end System;
...
```

Figure 8.3: Battery sensor model with two contracts specified on the `System` component.

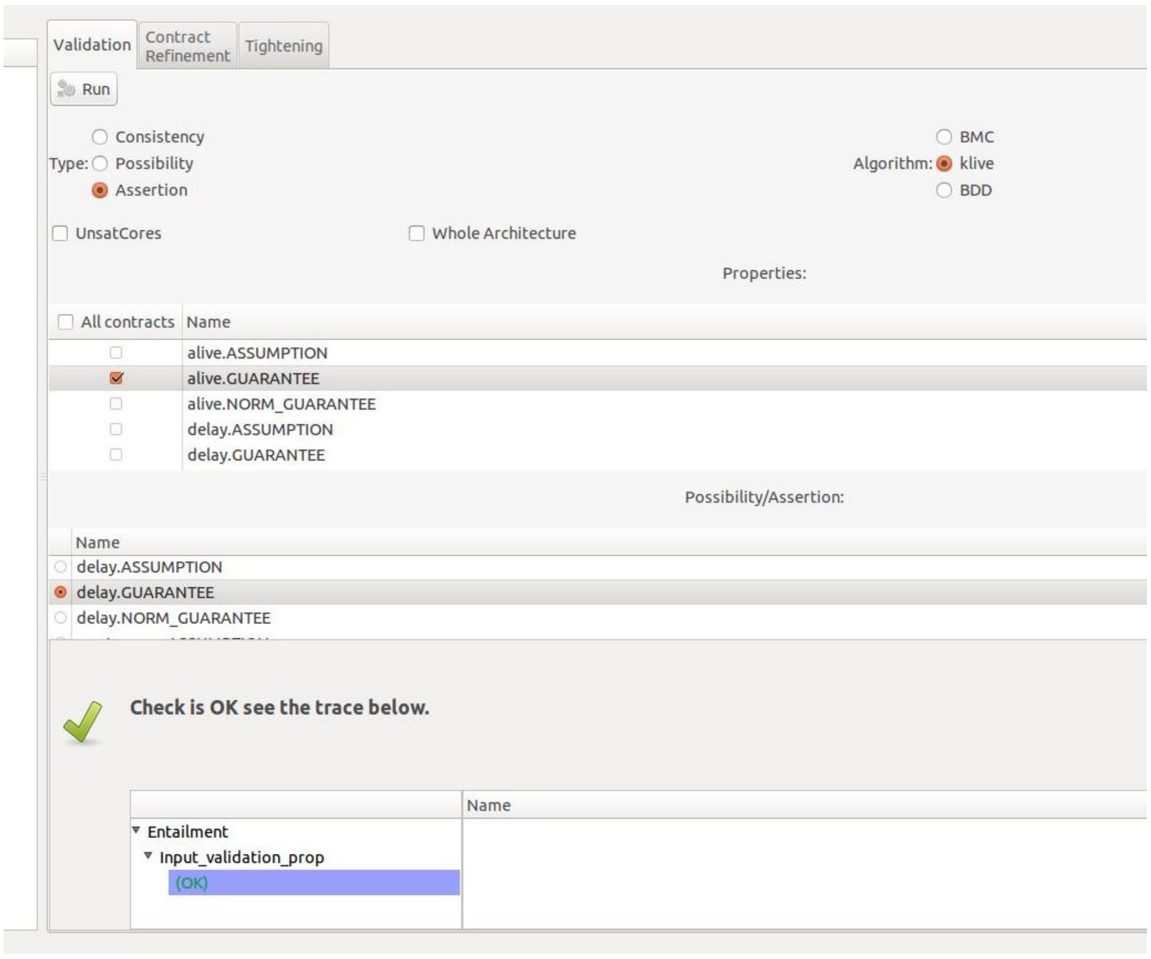
8.2 Contracts Validation

Once the contracts have been specified, we can validate them from the Validation GUI. The validation consists in a series of consistency or entailment checks, performed just to increase our confidence in the correct formulation of the properties in the contracts. For example, in the above example, we can check if the guarantee of the contract `alive` entails the guarantee of the contract `delay`, as one would expect since the latter is weaker. We do so as follows.

Steps

- Load the model `system_contracts.slim` as shown in Section 4.4.1
- Click on the *Validation* tab
- On the left, select the component `System`
- Click on the *Validation* sub-pane
- Select *Assertion* as type of check
- Select *klive* as algorithm (to have a proof since we do not expect counterexamples)
- Select `alive.GUARANTEE` as *Properties*
- Select `delay.GUARANTEE` as *Possibility/Assertion*
- Click the top-most *Run* button
- The tool reports that the assertion is true, as shown in Figure 8.4.

As additional check, we can verify that the contracts specified in the `System` component and its subcomponents are consistent. We do so as follows.



Type: Consistency / Possibility / [✓ Assertion]

Algorithm: BMC / [✓ klive] / BDD

☐ UnsatCores ☐ Whole Architecture

Properties:

☐ All contracts	Name
☐	alive.ASSUMPTION
☑	alive.GUARANTEE
☐	alive.NORM_GUARANTEE
☐	delay.ASSUMPTION
☐	delay.GUARANTEE

Possibility/Assertion:

	Name
○	delay.ASSUMPTION
●	delay.GUARANTEE
○	delay.NORM_GUARANTEE

Check is OK see the trace below.

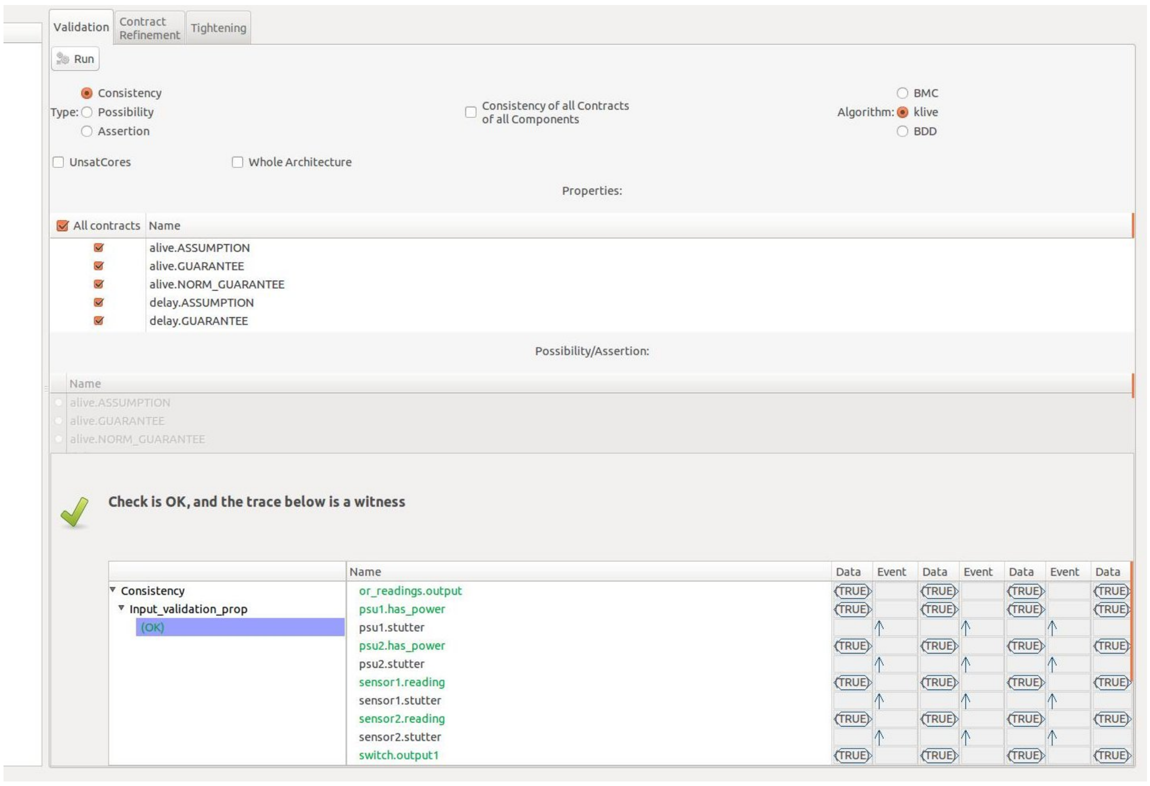
```
▾ Entailment
  ▾ Input_validation_prop
      (OK)
```

Figure 8.4: Proved that the property `alive.GUARANTEE` entails the property `delay.GUARANTEE` of the `system_contracts` example.

Steps

- Continuing from the previous steps, we already loaded the model `system_contracts.slim` and we are in the *Validation* tab, *Validation* sub-pane, for the `System` component
- Select *Consistency* as type of check
- Select *All Contracts* in the *Properties* pane
- Click the top-most *Run* button
- The tool provides a trace witnessing the consistency of the contracts as shown in Figure 8.5.

The *Validation* tab offers also the possibility of running a *Tightening* check; we refer to section 9.2.3 in the user manual for an explanation of this feature.



Validation | Contract Refinement | Tightening

Run

Type: [✓ Consistency] Possibility Assertion

☐ Consistency of all Contracts of all Components

Algorithm: BMC [✓ klive] BDD

☐ UnsatCores ☐ Whole Architecture

Properties:

✓ All contracts	Name
✓	alive.ASSUMPTION
✓	alive.GUARANTEE
✓	alive.NORM_GUARANTEE
✓	delay.ASSUMPTION
✓	delay.GUARANTEE

Possibility/Assertion:

Name
alive.ASSUMPTION
alive.GUARANTEE
alive.NORM_GUARANTEE

Check is OK, and the trace below is a witness

- Consistency
  - Input_validation_prop
    - (OK)

Name	Data	Event	Data	Event	Data	Event	Data
or_readings.output	TRUE		TRUE		TRUE		TRUE
psu1.has_power	TRUE		TRUE		TRUE		TRUE
psu1.stutter		↑		↑		↑	
psu2.has_power	TRUE		TRUE		TRUE		TRUE
psu2.stutter		↑		↑		↑	
sensor1.reading	TRUE		TRUE		TRUE		TRUE
sensor1.stutter		↑		↑		↑	
sensor2.reading	TRUE		TRUE		TRUE		TRUE
sensor2.stutter		↑		↑		↑	
switch.output1	TRUE		TRUE		TRUE		TRUE

Figure 8.5: Proved that the properties in all contracts of `System` component in the `system_contracts` are consistent.

8.3 Specification and Verification of a Contract Refinement

Once we validated the contracts, we can proceed by verifying that their refinement is correct. The refinement is specified in the model as shown in Figure 8.6.

We check the refinements as follows.

Steps

- Continuing from the previous steps, we already loaded the model `system_contracts.slim` and we are in the *Validation* tab
- Click on the *Contract Refinement* sub-pane
- Select *Add fairness assumption on component execution*
- Select *klive* as algorithm (to have a proof since we do not expect counterexamples)
- Click the top-most *Run* button
- The tool reports that the refinements are correct, as shown in Figure 8.7.

```
...
system implementation System.Imp
  subcomponents
...
  connections
...
  properties
    SLIMpropset::ContractRefinements => (
    [Contract => "alive";
    SubContracts => ("sensor1.reading", "sensor2.reading",
                     "psu1.power", "psu2.power",
                     "switch.switch", "or_readings.or_gate");
    ],
    [Contract => "delay";
     SubContracts => ("sensor1.reading", "sensor2.reading",
                      "sensor1.delay", "sensor2.delay",
                      "psu1.power", "psu2.power",
                      "psu1.delay", "psu2.delay",
                      "switch.switch", "or_readings.or_gate");
    ]);
end System.Imp;
...
```

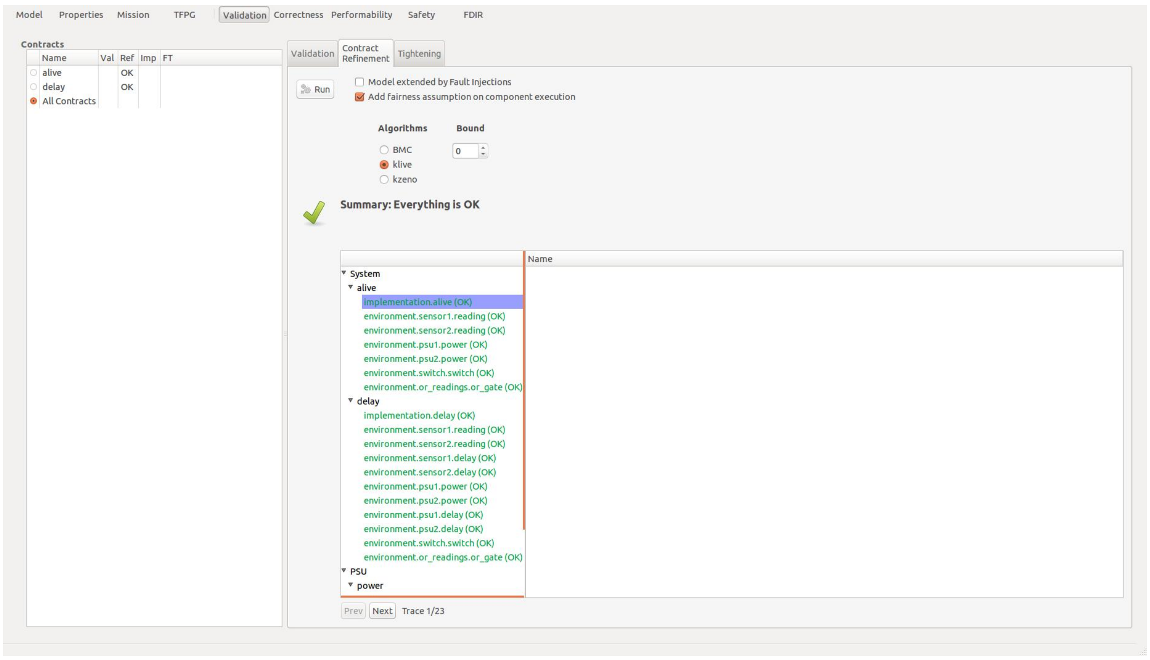
Figure 8.6: Contracts refinement in the `System` component implementation of the `system_contracts` model.

8.4 Generating Hierarchical Fault Tree from Contracts

Finally, we exploit the contracts to generate a hierarchical fault tree showing, at different levels of the refinement, the possible combination of failures subcomponents and environment (meant as not fulfillment of the contract guarantee and assumption) leading to the failure of the parent component. We do so as follows.

Steps

- Continuing from the previous steps, we already loaded the model `system_contracts.slim`
- Click on the *Safety* tab
- Click on the *Hierarchical Fault Tree Generation* sub-pane
- Select *Add fairness assumption on component execution*
- Select *klive* as algorithm
- Click the top-most *Run* button
- Once the analysis is completed the *View* button gets enabled; click it.



Contracts

Name	Val	Ref	Imp	FT
alive		OK		
delay		OK		
All Contracts				

[ ] Model extended by Fault Injections
[✓] Add fairness assumption on component execution

Algorithms: BMC [✓ klive] kzeno
Bound: 0

Summary: Everything is OK

```
▼ System
  ▼ alive
      implementation.alive (OK)
      environment.sensor1.reading (OK)
      environment.sensor2.reading (OK)
      environment.psu1.power (OK)
      environment.psu2.power (OK)
      environment.switch.switch (OK)
      environment.or_readings.or_gate (OK)
  ▼ delay
      implementation.delay (OK)
      environment.sensor1.reading (OK)
      environment.sensor2.reading (OK)
      environment.sensor1.delay (OK)
      environment.sensor2.delay (OK)
      environment.psu1.power (OK)
      environment.psu2.power (OK)
      environment.psu1.delay (OK)
      environment.psu2.delay (OK)
      environment.switch.switch (OK)
      environment.or_readings.or_gate (OK)
▼ PSU
  ▼ power
```

Name

Trace 1/23

Figure 8.7: Proved that the contract refinements of the `system_contracts` example are correct.

- The tool reports two fault trees, one for each property. In Figure 8.8 is shown the one of the contract `alive`.

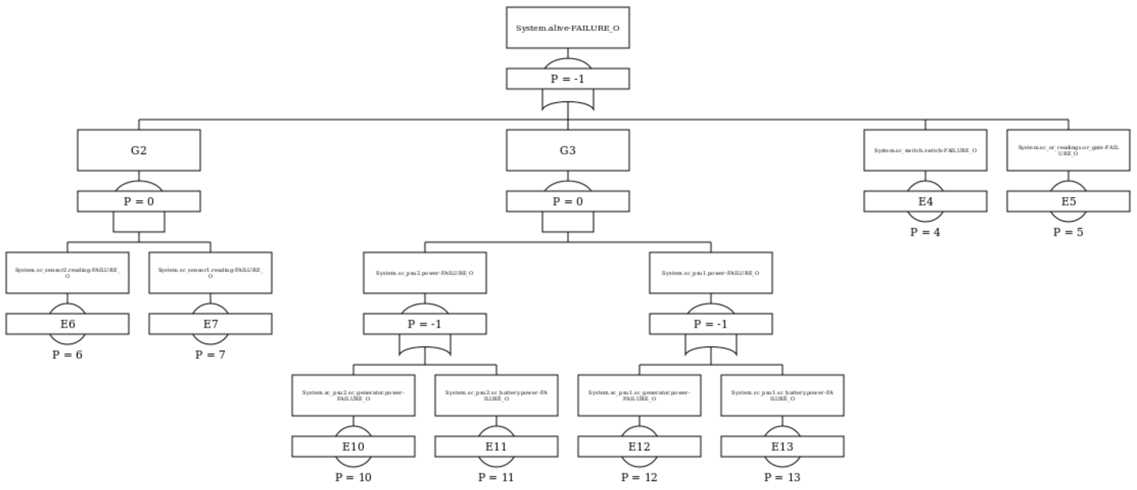

System.alive-FAILURE_O
P = -1
G2
P = 0
G3
P = 0
System.sc_switch.switch-FAILURE_O
E4
P = 4
System.sc_or_readings.or_gate-FAILURE_O
E5
P = 5
System.sc_sensor2.reading-FAILURE_O
E6
P = 6
System.sc_sensor1.reading-FAILURE_O
E7
P = 7
System.sc_psu2.power-FAILURE_O
P = -1
System.sc_psu1.power-FAILURE_O
P = -1
System.sc_psu2.sc_generator.power-FAILURE_O
E10
P = 10
System.sc_psu2.sc_battery.power-FAILURE_O
E11
P = 11
System.sc_psu1.sc_generator.power-FAILURE_O
E12
P = 12
System.sc_psu1.sc_battery.power-FAILURE_O
E13
P = 13

Figure 8.8: Hierarchical Fault Tree generated from the contract refinements of `system_contracts`.

Chapter 9

Hints and Tips

This section contains a list of recommendations that will help the user make the best of the COMPASS toolset and address the most common issues and pitfalls. It covers modeling in SLIM, running the toolset, and perform analyses using COMPASS.

9.1 Modeling

Avoid deadlocks A deadlock is a state in a model specification, that does not have any outgoing transition. Deadlocks should be avoided, since they may cause verification results that are not trustable, for instance in model checking (compare Section 9.3). The presence of deadlocks in a specification may indicate a behavior that was not intended by the modeler (e.g., a missing transition in a transition system, or an inconsistency between transitions of different components that move synchronously). Deadlock checking analysis can be used to inspect a model for deadlocks.

Avoid Zeno cycles and timelocks A time lock is an example of deadlock that may arise in timed models, as a consequence of inconsistencies in mode triggers, invariants or guards, or as a result of communication of different components. A Zeno cycle refers to system computations involving an infinite number of discrete (i.e., not timed) transitions steps within a bounded period of time. Both Zeno cycles and timelocks should be avoided, since they may cause verification results that are not trustable. COMPASS offers Zeno analysis to check a model for zenoness. Moreover, Section 5.1.1 of the user manual contains modeling advice and sufficient conditions to avoid Zeno cycles and timelocks.

Always bound clocks Clocks that grow arbitrarily large should be avoided, whenever possible, since they may complicate verification. For instance, in fair model checking, the model checker looks for counterexamples that may be extended to an infinite fair path, and can be represented using loops. A clock that grows arbitrarily prevents the existence of such counterexamples. The modeler should ensure that clocks are reset at least once on each cycle of a transition system (compare Section 5.1.1. in the user manual). COMPASS also offers Time Divergence analysis to check whether all clocks are bounded.

9.2 Running the Toolset

Quick start Section 3 of the user manual explains in detail how to obtain a copy of the toolset and the pre-requisites and instructions about how to install it. COMPASS is also distributed, for convenience, pre-installed on a self-contained virtual machine. Section 4.3 in this document provides a quick start about how to run the toolset using the GUI.

Using the command-line interface In addition to the GUI, COMPASS also comes with a command-line interface that can be used to run batch jobs. The command-line interface is based on a set of python scripts that expose the verification capabilities of the toolset (several options are available). The command-line scripts are explained in detail in Appendix A of the user manual.

Interpreting errors and error messages Error messages are provided either in the main window of the COMPASS toolset (see "Output Console", tabs "Compiler" and "Logging") or in contextual sub-windows or tabs that are local to specific analyses, or in pop-up windows. Generally, error messages are self-explanatory, and they are documented in the user manual. Compiler errors are often due to syntactic errors in models, or semantic rules that are violated; the latter are usually reported along with a pointer to the syntax/semantics document where they are documented. The logging provides a low-level report of the calls that have been made to the underlying solving engine; pointers to console output produced by this engines are included, however their inspection requires expertise. Bugs, undocumented error messages and other run-time errors (e.g., run-time exceptions) may be reported to the COMPASS staff using the channels described in Section 10 of the user manual.

Using time and memory limits It is possible to limit the resources (time and memory) that are available to run analysis tasks. It the time and/or memory bound is exceeded, the analysis is terminated automatically. The "Processes Monitor" can be run from the *View* menu, item *Processes*, and it can be used to view the set of running processes and set time and/or memory limits.

9.3 Analysis

The following provides some hints towards to use of various COMPASS analysis tools. In particular, some if the parameters that need to be set by the user are discussed here. Aside form these hints, more details can be found in Section 9 in the user manual.

Selecting a verification engine COMPASS provides different verification engines, that often are provided for the same activity. For instance, model checking can be executed with the BDD-based engine, or the SAT engines (BMC, klive). The same applies to other activities, such as fault tree generation, which also provided the SMT-based ParamIC3 engine. Some engines may have restrictions, for instance the BDD-based engine is only available for discrete models. Other engines such as SAT may be inconclusive or may perform verification up to a given bound (e.g., "The property is true up to bound . . . "). Different engines may have different performances. The engine that often performs best in practice is selected as default engine, but other engines may be tried. We refer to Section 9.4.4 of the user manual for an in-depth discussion of the pros and cons of the different engines. For the selection of the SAT bound, see also next item.

Choosing a verification bound The SAT engine provides the BMC-based and klive-based verification algorithms. We refer to Section 9.4.4 of the user manual for a more detailed explanation. Choosing a bound may impact the outcome of the verification task. A "...true up to bound ..." outcome means that the verification engines only explored the state space up to the given bound – corresponding to the maximum length of the execution traces that have been explored. Choosing a higher bound may yield more accurate results, but may also increase the analysis time. Sometimes, increasing the bound may allow the engine to complete the analysis and produce a conclusive result. In general, there is no rule of thumb in choosing the SAT bound. It is useful to run simulations of the model under analysis, in order to understand how long are the execution traces that are to be analyzed: the SAT bound should be as long as the length of such traces. Similar considerations hold for the bounds for the klive routines – more details can be found in Section 9.4.4 of the user manual.

Probabilistic Analysis versus Simulation For performability, both IMC analysis and Model simulation are available, however both have their specific use cases. First and foremost, the meaning of their output differs: whereas for analysis the value is guaranteed to be the actual probability, for simulation an error is possible (the magnitude of which can be controlled). However, analysis can be much more expensive to perform, and in some cases is not possible – in particular for timed/hybrid models, limiting the choice to simulation. Therefore, the recommendation is to use analysis for models that are not too large, or for which the results have to be highly accurate, and simulation for those where this is not possible or required. In particular, quick prototyping can be performed more easily with simulation.

Performability Analysis versus Fault Tree Evaluation Both performability analysis and fault tree evaluation strive towards the same goal: quantifying the reliability of the system. The main difference between the two is that the prior calculates this on the input model directly, whereas the latter does so via an abstraction in the form of a fault tree. Ideally the two match, but this is not always true. Fault trees lend themselves very well to describe the possible propagation of faults in the model and make their effect much easier to understand, but cannot capture the full behavioral aspects of the model, something that performability analysis does take into account. Therefore, if simply the reliability needs to be known, performability analysis will provide the closest representation, but for better understanding of the model, a fault tree provides a better representation.

Model-Based vs. Contract-Based Fault-Tree Generation Fault trees can be generated either by specifying the error model and fault-injection or by specifying the contract refinement. The results are complementary as the first focused on the faults manually specified by the user, while the second has a predefined notion of failure, i.e., the inability of a component to ensure the guarantee or the inability of a component environment to ensure the assumption. The first produces a flat fault tree, while the second produces a hierarchical fault tree structured along the architectural decomposition. The user may be interested in performing both or either of the analyses. The choice depends on the availability of the fault injection (for the flat fault tree) and the contract specification (for the hierarchical fault tree).

Bibliography

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

[2] SLIM 3.0 - Syntax and Semantics. Technical report, COMPASS Consortium, 2016. Version 1.2.

[3] COMPASS Toolset User Manual. Technical report, COMPASS Consortium, 2016.

[4] Space engineering: System engineering general requirements. ECSS Standard E-ST-10C, European Cooperation for Space Standardization, March 2009.

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

[6] Space product assurance: Availability analysis. ECSS Standard Q-ST-30-09C, European Cooperation for Space Standardization, July 2008.

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

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

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

[10] Architecture Analysis and Design Language (AADL) V2. SAE Draft Standard AS5506 V2, International Society of Automotive Engineers, March 2008.

[11] Architecture Analysis and Design Language Annex (AADL), Volume 1, Annex E: Error Model Annex. SAE Standard AS5506/1, International Society of Automotive Engineers, June 2006.