Under consideration for publication in Formal Aspects of Computing

Model-based Safety Assessment of a Triple Modular Generator with xSAP

Marco Bozzano[1], Alessandro Cimatti[1], Marco Gario[1], David Jones[2], Cristian Mattarei[1]

[1]Fondazione Bruno Kessler, Trento (Italy)
[2] Bellevue, WA (USA)

Abstract. The system design process needs to cope with the increasing complexity and size of systems, motivating the replacement of labor intensive manual techniques with automated and semi-automated approaches. Recently, formal methods techniques, such as model-based verification and safety assessment, have been increasingly used to model systems under fault and to analyze them, generating artifacts such as fault trees and FMEA tables.

In this paper, we show how to apply model-based techniques to a realistic case study from the avionics domain: a high integrity power distribution system, the Triple Modular Generator (TMG). The TMG is composed of a redundant and reconfigurable plant and a controller that must guarantee a high level of reliability. The case study is a significant challenge, from the modeling perspective, since it implements a complex reconfiguration policy, specified via a number of requirements in natural language, including a set of mutually dependent and potentially conflicting priority constraints. Moreover, from the verification standpoint, the controller must be able to handle an exponential number of possible faulty configurations.

Our contribution is twofold. First, we formalize and validate the requirements and, using a constraint-based modeling style, we synthesize a correct by construction controller, avoiding the enumeration of all possible fault configurations, as is currently done by manual approaches. Second, we describe a comprehensive methodology and process, supported by the xSAP safety analysis platform that targets the modeling and safety assessment of faulty systems. Using xSAP, we are able to automatically extract minimal cut sets for the TMG. We demonstrate the scalability of our approach by analyzing a parametric version of the TMG case study that contains more than 700 variables and 90 faults.

Keywords: Power Generation Systems, Model Checking, Model-Based Safety Assessment, Fault Tree Analysis, Minimal Cut Sets.

1. Introduction

Safe operation of complex systems, in many industrial domains, relies on the ability to provide continuous and reliable electrical power. For instance, in avionics, reliable electrical generators are essential to ensure safe operation of aircraft. Power systems are subject to critical requirements, since they have to ensure correct operation even in non-nominal

Correspondence and offprint requests to: Marco Bozzano, Fondazione Bruno Kessler, Via Sommarive 18, 38123 Trento, Italy. e-mail: bozzano@fbk.eu

situations (in presence of faults). This has led to the adoption of complex redundant architectures (for the plant and for the controller), resulting in a significant increase in the complexity of their analysis and safety assessment. The More Electric Aircraft (MEA) concept [Ray18] has made all these issues even more serious.

Similar remarks apply to the electrical power industry. Distribution systems are designed for redundancy, e.g. they are connected to multiple power sources and they exploit redundant paths to guarantee reliability/ availability of power delivery to customers. Recent concepts include the use of the distributed generation sources and "self-healing" grid [ea10]. Microgrids [KIHD08] provide an additional example of systems that need to implement complex design requirements. Their architecture typically supports a fine-grained load management, local control of power generation, and the possibility to operate connected or disconnected from the main grid. The complexity of the design of these systems requires sophisticated capabilities to monitor and control the power sources and the loads, and to effectively diagnose and recover from faults (e.g., by isolating faulty parts of the grid).

The design, verification, safety assessment and operational control of these systems is a highly labor intensive and error-prone activity, encouraging automated and formally-verifiable techniques. The growing interest of industry in applying formal methods is justified by the growing complexity of the systems to be analyzed, and by the high degree of assurance that is required for their deployment. Recent trends in this areas include the application of the Model-Based Safety Assessment (MBSA) paradigm [BVÅ03, JMWH05, BV10]. In MBSA, design and safety engineers share the same system models. Faulty behaviors can be automatically injected into the nominal models, and the resulting extended models can be analyzed automatically by means of exhaustive techniques, e.g. based on model checking [BCC$^+$03, BCK$^+$14, BCP$^+$15].

In this paper we demonstrate the application of MBSA to a realistic industrial case study: the TMG (Triple Modular Generator). The TMG is a redundant on-board power distribution system (including generators, buses and circuit breakers) that implements a complex reconfiguration policy, whose goal is to provide continuous power to buses, insofar as possible. The case-study is composed of a plant representation and a set of informal, complex requirements that express the expected behavior of the overall system (plant, controller). Specifically, a complex priority scheme constrains the available powering options (alternative generator-bus paths) that can be used by the controller to achieve the goal. Furthermore, priority requirements are mutually dependent, namely powering options for different buses may be conflicting and need to be resolved. The case-study was chosen to be comparable in complexity to the high voltage power systems of many previous generation aircraft, although details of the architecture may vary. For such systems, the control strategy could be manually synthesized by enumerating all the possible faulty configurations (determined by the possible fault combinations of the different components), however this approach would become rapidly unmanageable, due to the exponential number of such configurations.

The purpose of this work is to describe a comprehensive approach, based on formal methods, that can be applied to the modeling of the system, the formalization of the requirements and the formal verification and validation of the system characteristics. The approach is supported by the XSAP toolset [BBC$^+$16, xSA19], which provides a broad set of techniques covering all of the steps of formal modeling and assessment. Using XSAP, we are able to verify and validate the requirements expressing the expected behavior of the system, synthesize a formal implementation of the controller that meets such requirements, and evaluate its reliability. The synthesis of the controller relies on a constraint-based formalization, which avoids the explicit enumeration of the faulty configurations.

Our contribution is twofold. First, we model and fully analyze a significant case study. The modeling of the system, in particular of the controller priority scheme, is non-trivial. We thoroughly explain how to model prioritized requirements and control laws for a closed loop system with potentially conflicting conditions. Our style of modeling is very general, and can be adapted to similar case studies. Moreover, the case study we present has size 3 (namely, there are 3 buses, 3 generators, and 6 circuit breakers) for explanatory purposes, but the experiments are carried over to way larger models, up to size 18 (the latter model has more than 700 variables and 90 possible faults). The second contribution is a comprehensive presentation of the XSAP process and tool, along with the underlying techniques. The automated extraction of results is made possible and effective thanks to the scalability of the mature techniques implemented in the tool, namely fault tree analysis via parameter synthesis and anytime computation [BCMG15].

The rest of this paper is organized as follows. In Section 2 we present some background. In Section 3 we present the TMG case study. In Section 4 we present the XSAP process and tool. In Section 5 we illustrate the modeling of the TMG. In Section 6 we discuss formal verification and safety assessment. In Section 7 we discuss diagnosability and fault detection and identification analysis. In Section 8 we describe the experimental evaluation. In Section 9 we discuss related work. Finally, in Section 10 we draw some conclusions and discuss future work.

2. Background

In this section we introduce some notions that we use in the rest of the paper. First, we introduce the notion of symbolic transition system, which we use to represent systems and their evolution (both with and without faults). Symbolic transition systems are amenable to formal verification using tools such as XSAP. We introduce temporal logic, which is used to express properties over transition systems, and symbolic model checking.

2.1. Symbolic Transition Systems

In this paper we represent systems symbolically as *symbolic transition systems* (STS).
An STS is a tuple $S = \langle V, V_o, W, W_o, F, I, T\rangle$, where:

- V is the set of state variables;
- $V_o \subseteq V$ is the set of observable state variables;
- W is the set of input variables;
- $W_o \subseteq W$ is the set of observable input variables;
- $F \subseteq W$ is the set of faults;
- I is a formula over V defining the initial states;
- T is a formula over V, W, V' (with V' being the next version of the state variables) defining the transition relation.

In the context of this paper, we assume that variables range over finite domains[1]. A *state* s is an assignment to the state variables V. We denote with s' the corresponding assignment to V'. An *input* i is an assignment to the input variables W. Given a state s and a Boolean variable $p \in V$, we write $s \models p$ if $s(p)$ is assigned to true. Given a state s and a set of variables V, we denote with $s_{|V}$ the restriction of s to the variables in V; similarly for inputs.

The *observable part* $obs(s)$ of a state s is the projection of s on the subset V_o of observable state variables. The observable part $obs(i)$ of an input i is the projection of i on the subset W_o of observable input variables. Thus, $obs(s) = Proj(s, V_o)$ and $obs(i) = Proj(i, W_o)$. A *trace* of S is an infinite sequence (i.e., a path) $\pi = s_0, i_1, s_1, i_2, s_2, \ldots$ of states and inputs such that s_0 satisfies I and, for each $k \geq 0$, $\langle s_k, i_{k+1}, s_{k+1}\rangle$ satisfies T. We write Π_S for the set of traces of S. The observable part of π is $obs(\pi) = obs(s_0), obs(i_1), obs(s_1), obs(i_2), obs(s_2), \ldots$. Given a trace $\pi = s_0, i_1, s_1, i_2, s_2, \ldots$ and an integer $k \geq 0$, we denote with π^k the finite prefix $s_0, i_1, \ldots, s_k$ of π containing the first $k+1$ state-input pairs. We denote with $\pi[k]$ the $k+1$-th state s_k. We say that s is *reachable* (in k steps) in S if and only if there exists a trace $\pi \in \Pi_S$ such that $s = \pi[k]$ for some $k \geq 0$. We say that the transition relation is *total* if and only if for each state s there exists a successor state for some input i, that is, $\forall s \exists i, t$ such that $T(s, i, t)$ is valid. Without loss of generality, in the following we assume that a system is total, and consider infinite traces only. We say that S is deterministic if there are no two initial states s_0 and s'_0 s.t. $obs(s_0) = obs(s'_0)$, and there are no two transitions $\langle s, i_1, s'_1\rangle$ and $\langle s, i_2, s'_2\rangle$ from a reachable state s s.t. $obs(i_1) = obs(i_2)$ and $obs(s'_1) = obs(s'_2)$.

Let $S^1 = \langle V^1, V^1_o, W^1, W^1_o, F^1, I^1, T^1\rangle$ and $S^2 = \langle V^2, V^2_o, W^2, W^2_o, F^2, I^2, T^2\rangle$ be two transition systems with $\emptyset = (V^1 \backslash V^1_o) \cap V^2 = V^1 \cap (V^2 \backslash V^2_o) = (W^1 \backslash W^1_o) \cap W^2 = W^1 \cap (W^2 \backslash W^2_o)$. We define the *synchronous composition* $S^1 \times S^2$ as the transition system $\langle V^1 \cup V^2, V^1_o \cup V^2_o, W^1 \cup W^2, W^1_o \cup W^2_o, F^1 \cup F^2, I^1 \wedge I^2, T^1 \wedge T^2\rangle$.

2.2. Temporal Logic

In this paper we use Linear Temporal Logic (LTL) extended with past operators and Computation Tree Logic (CTL) [Pnu77, LMS02, LPZ85, AE90, Var01] .

A formula in LTL over variables V is defined as

$$\beta ::= p \mid \beta \wedge \beta \mid \neg\beta \mid O\beta \mid Y\beta \mid G\beta \mid F\beta \mid X\beta \mid \beta U \beta$$

where p is a propositional variable. Given a trace $\pi = s_0, i_1, s_1, i_2, s_2, \ldots$ and an index i, the semantics of LTL is defined on linear traces as follows:

- $\pi, i \models p$ iff $s_i \models p$;
- $\pi, i \models \beta_1 \wedge \beta_2$ iff $\pi, i \models \beta_1$ and $\pi, i \models \beta_2$;

[1] Without loss of generality, we can assume that variables are Boolean, since finite-domain variables can be encoded using Boolean ones.

- $\pi, i \models \neg\beta$ iff $\pi, i \not\models \beta$;
- Once: $\pi, i \models O\beta$ iff $\exists j \leq i.\ \pi, j \models \beta$;
- Yesterday: $\pi, i \models Y\beta$ iff $i > 0$ and $\pi, i-1 \models \beta$;
- Globally: $\pi, i \models G\beta$ iff $\forall j \geq i.\ \pi, j \models \beta$;
- Finally: $\pi, i \models F\beta$ iff $\exists j \geq i.\ \pi, j \models \beta$;
- Next: $\pi, i \models X\beta$ iff $\pi, i+1 \models \beta$;
- Until: $\pi, i \models \beta_1 U \beta_2$ iff there exists $j \geq i$ such that $\pi, j \models \beta_2$ and for all k, $i \leq k < j$, $\pi, k \models \beta_1$.

Given an LTS $S = \langle V, V_o, W, W_o, F, I, T\rangle$, we write $S \models \beta$ if and only if for every trace π of S, $\pi, 0 \models \beta$. Notice that $Y\beta$ is always false in the initial state, and that we use a reflexive semantics for the operators U, F, G, S and O. We use the abbreviations $Y^n\beta = YY^{n-1}\beta$ (with $Y^0\beta = \beta$), $O^{\leq n}\beta = \beta \vee Y\beta \vee \cdots \vee Y^n\beta$ and $F^{\leq n}\beta = \beta \vee X\beta \vee \cdots \vee X^n\beta$.

CTL is similar to LTL, but every temporal operator is preceded by a path quantifier. A formula in CTL over variables V is defined as

$$\beta ::= p \mid \beta \wedge \beta \mid \neg\beta \mid EG\beta \mid EF\beta \mid EX\beta \mid E\beta U\beta \mid$$

where p is a propositional variable.

CTL properties are evaluated over states(i.e., over the trees that start from such states). Formally, Given an LTS $S = \langle V, V_o, W, W_o, F, I, T\rangle$ and a state s:

- $S, s \models p$ iff $s \models p$;
- $S, s \models \beta_1 \wedge \beta_2$ iff $S, s \models \beta_1$ and $S, s \models \beta_2$;
- $S, s \models \neg\beta$ iff $S, s \not\models \beta$;
- Exists Globally: $S, s \models EG\beta$ iff there is a path $\pi = s_0, i_1, s_1, i_2, s_2, \ldots$ such that $s_0 = s$ and for all $j \geq 0$, $S, s_j \models \beta$;
- Exists Finally: $S, s \models EF\beta$ iff there is a trace $\pi = s_0, i_1, s_1, i_2, s_2, \ldots$ such that $s_0 = s$ and for some $j \geq 0$, $S, s_j \models \beta$;
- Exists Next: $S, s \models EX\beta$ iff there is a trace $\pi = s_0, i_1, s_1, i_2, s_2, \ldots$ such that $s_0 = s$ and $S, s_1 \models \beta$;
- Exists Until: $S, s \models E\beta_1 U \beta_2$ iff there is a path $\pi = s_0, i_1, s_1, i_2, s_2, \ldots$ such that $s_0 = s$ and for some $j \geq 0$, $S, s_j \models \beta_2$ and for all $0 \leq k < j$, $S, s_k \models \beta_1$.

The universal quantifier A can be defined as an abbreviation, e.g. $AG\beta := \neg EF\neg\beta$.

2.3. Symbolic Model Checking

Given a (symbolic) transition system S and a temporal logic formula β, model checking is an automated, formal verification technique whose goal is to verify whether the system S satisfies the specification β, written $S \models \beta$. Model checking implements this verification by performing an exhaustive exploration of the state space of the given system, thus covering every possible system behavior. Termination of the model checking routines is guaranteed in case of finite state models. Model checking also provides the designer with useful information in case a specification is not satisfied, namely it generates a counterexample trace witnessing the violation.

Traditionally, model checking has been approached in different ways, in particular older routines are based on the explicit representation of the states of system (so-called *explicit state model checking*). A more recent advance is the introduction of *symbolic model checking* [McM93]. In this approach, sets of states and transitions are represented and manipulated symbolically, using efficient logical representations, such as Ordered Binary Decision Diagrams [Bry92] (BDDs). An even more recent advance is the use of SAT solvers and the introduction of bounded model checking (BMC) [BCRZ99, Bra11, CGMT13].

Examples of state-of-the-art model checkers are NuSMV [CCGR00, NuS19] and nuXmv [CCD+14, nuX19].

The xSAP platform builds upon the nuXmv model checker, inheriting its model checking capabilities for verification and validation, while implementing specialized routines for safety assessment, based on extensions of model checking. Both BDD-based and SAT-based routines are available. In particular, in this paper we use the IC3-based routines described in [BCMG15]. Diagnosability, Fault Detection and Identification capabilities are based on the framework described in [BCGT14, BCGT15] and are also implemented using techniques based on model checking.

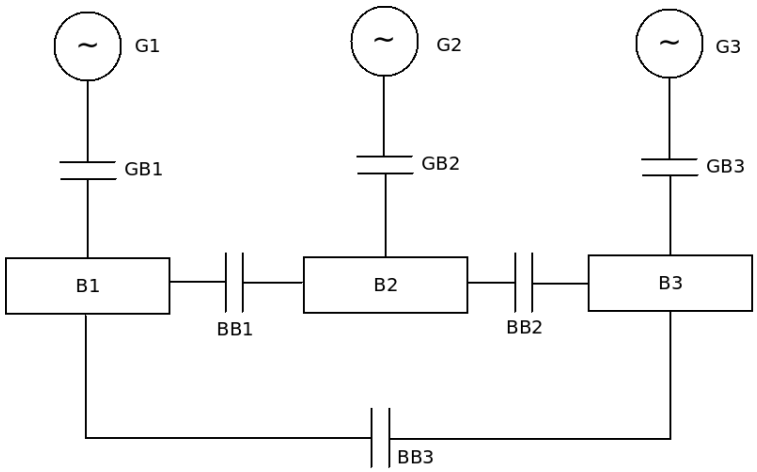

Fig. 1. Triple Modular Generator.

ID	Description
R1	No bus shall be connected to more than one power source at any time.
R2	If any power source is on, then all buses will be powered.
R3	Bus power source priority and source to bus path priority schemes shall be respected at all times (see Fig. 3 and 4).
R4	If no power source is on, then all buses will be unpowered.
R5	Any single/dual component failure shall not cause other system requirements to be violated.
R6	Never more than two generators are on, unless required in case of failures.

Fig. 2. TMG Requirements.

3. The Triple Modular Generator (TMG)

The Triple Modular Generator (TMG) is a redundant on-board power distribution system. Its complexity is comparable to the high voltage power systems of many current generation aircraft, although the specific architecture may vary.

The TMG system, illustrated in Fig. 1, is composed of:

- *Generators* (G1, G2 and G3): they act as power sources, they have an output connection and may have two possible states: ON (the generator provides AC power to the output connection) and OFF (the generator is switched off).
- *Circuit Breakers* (GB1, GB2, GB3, BB1, BB2 and BB3): they have an input and an output connection and may have two possible states: OPEN (the circuit is open, i.e. the input connection is isolated from the output connection) and CLOSED (the circuit is closed, i.e. the input connection is electrically connected to the output connection).
- *Buses* (B1, B2 and B3): they represent the loads of the system. They receive power from one of their input connections, and they propagate it to the other two ports.

Moreover, a *Controller* (not shown in Fig. 1) is connected in closed-loop with the system, and it is responsible for monitoring the system and carrying out re-configuration, if needed due to the presence of faults. To this aim, it can send commands to generators and circuit breakers.

In the nominal case (no faults) only one generator is active, and all buses are powered. The TMG system has several degrees of redundancy (triple redundancy scheme): backup generators may take over as power sources, in case of failure of the primary generator; moreover, each bus may be powered using alternate power connections, in case of failure of circuit breakers. Requirements are imposed on the controller, prescribing priorities on the way buses should be fed (i.e., on the bus and power line to be used).

Faults of the TMG The TMG components may fail in the following ways:

- Generators can fail *off* permanently ('stuck-at-off' fault). In this case, a generator does not provide output power.

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

Fig. 3. Bus power source priority.

Paths	Priority	B1	B2	B3
G1	High	—	BB1	BB3
	Low	—	BB3-BB2	BB1-BB2
G2	High	BB1	—	BB2
	Low	BB2-BB3	—	BB1-BB3
G3	High	BB3	BB2	—
	Low	BB2-BB1	BB3-BB1	—

Fig. 4. Source to bus path priority.

- Circuit breakers can fail *open* or *closed*, possibly transiently ('stuck-at-open' and 'stuck-at-closed' faults). In this case, the corresponding circuit is, respectively, open or closed.
- Buses may fail (short circuit) if they are powered simultaneously from two (or more) input connections. In this case, a bus becomes broken permanently.

Faults of generators and circuit breakers are primitive faults (random faults due to, e.g., wearing out), whereas bus faults are induced faults (e.g., due to controller errors). A requirement is imposed on the controller to avoid bus shorts.

The controller can monitor the state of the components using some sensors, and can send commands to the generators (*switch-on*, *switch-off*) and to the circuit breakers (*open*, *close*).

Requirements of the TMG Controller The goal the TMG controller is to ensure, insofar as possible, that all the buses are always powered, even in presence of faults. To achieve this goal, the controller monitors the system and can send commands to the generators and the circuit breakers, as appropriate. The controller has also to satisfy a set of additional requirements. The full list of requirements is shown in Fig. 2, and their intended purpose is described as follows:

R1 avoids bus shorts, which occur when a bus is connected simultaneously to more than one power source.

R2 states that one generator is sufficient to power all the buses. Notice that this requirements may be violated in case of multiple faults.

R3 prescribes priorities on the ways buses should be powered, namely which generator and which path (power line) should be used.

R4 expresses that the generators are the only source of power in the system.

R5 requires the system to be resilient to single and double faults.

R6 states that at most two generators should be on at any time unless, due to faults, three generators are necessary (to power the buses).

Bus power source priority and source to bus path priority schemes are described in Fig. 3 and Fig. 4. For instance, bus B1 (first row in Fig. 3) must be powered with highest priority using generator G1, if not possible using G2 (medium priority) or G3 (lowest priority). When powered by G1, bus B1 should be powered directly (i.e., using GB1, indicated as —, see first row in Fig. 4); when powered by G2, B1 should be powered with highest priority using the path going through (GB2 and) BB1, and with lowest priority using the path through (GB2 and) BB2-BB3. The remaining cases are similar.

4. Modeling and Analysis Process Using xSAP

In this section we discuss the modeling and analysis process supported by xSAP.xSAP is platform for Model-Based Safety Assessment [BBC+16, xSA19], implementing a variety of modeling and analysis techniques that automate activities that are traditionally performed manually. Specifically, supported techniques include, among others:

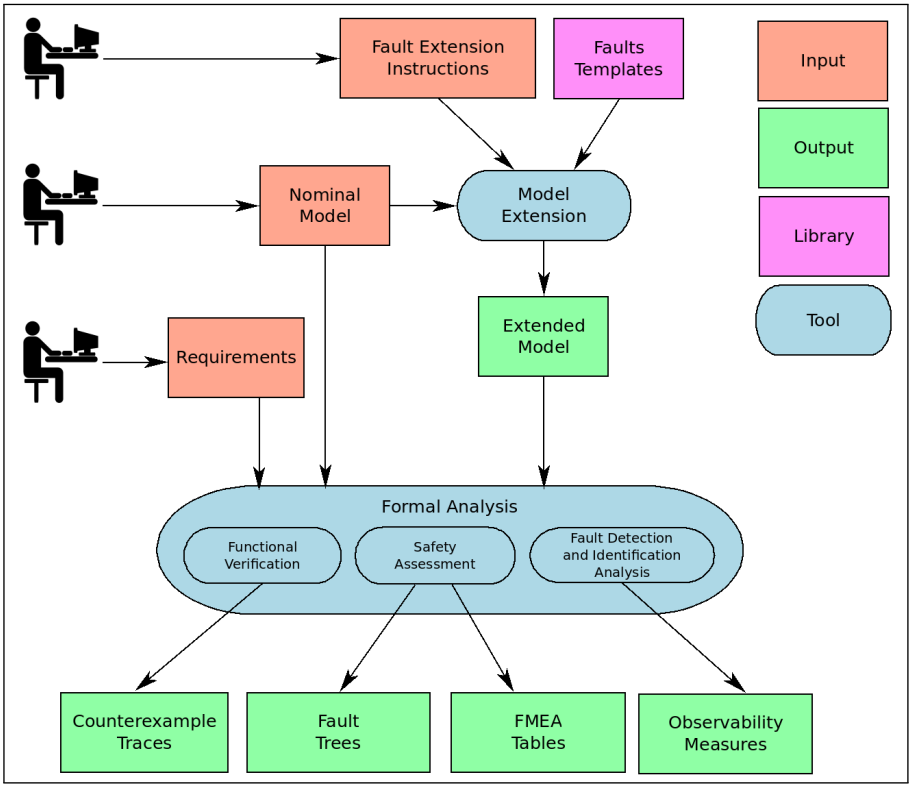

Fig. 5. XSAP Modeling and Analysis Process.

- *Definition of faults* and *Model Extension*: automatic extension of a system model with the definition of fault modes. Typical fault modes (e.g., 'stuck_at') can be defined for reuse in a common library, and system components can be independently annotated for extension, to include one or more fault modes.
- *Fault Tree Analysis (FTA)*: automated generation of *minimal cut sets* and *fault trees* from an extended model.
- *Failure Modes and Effects Analysis (FMEA)*: automated generation of FMEA tables from an extended model.
- *Common Cause Analysis*: analysis of events that may invalidate the hypothesis of independence of faults (e.g., cascading faults).
- *Diagnosability Analysis*, *Fault Detection and Identification Analysis*: evaluation of the adequateness of the level of observability (e.g., sensor allocation) of a system for the purpose of diagnosing faults, and the adequateness of an existing diagnoser.

4.1. xSAP Process

The XSAP process is illustrated in Fig. 5. It supports a unifying methodology that covers both modeling and verification of safety critical systems. The main steps of the process are:

Modeling nominal behavior The *nominal model* describes the behavior of a given system, when everything works as expected (absence of faults). The model can be written using the provided editors, and is an input for the following phases of model extension and functional verification. The model is written in the SMV language.

Model extension Model extension is performed to enrich the nominal model with the specification of the possible faults, that may affect the behavior of the system. The result of model extension is called *extended model* (i.e., nominal model extended with faulty behaviors). Model extension can be performed either manually or automatically. In the first scenario, the model is written manually, and XSAP provides primitives to declare *fault variables* (i.e., input variables controlling the triggering of faults) in the extended model. In the second scenario, model extension is carried out automatically by XSAP, taking as input some *fault templates* (retrieved from a *fault library*),

and *fault extension instructions* (i.e., directives that specify how to instantiate the fault templates and how to link the faults with their effects on the nominal model). Similarly to the nominal model, the extended model is written in the SMV language.

Modeling requirements The goal of this step is the elicitation and specification of system requirements. Different typologies of requirements may be covered, e.g. functional requirements, safety requirements, performance requirements, etc.

Functional verification Functional verification has the goal to assess the operational correctness of a given system. It consists in verifying that the system will operate correctly with respect to a set of requirements. Functional verification can be performed both on the nominal model and on the extended model (to assess the behavior of the system in nominal conditions or in presence of faults).

Safety assessment Safety assessment has the goal to assess the robustness of a given system in presence of faults. Safety assessment of critical systems is typically performed in parallel with system design, to ensure that the system meets the safety requirements that are necessary for its deployment and use. The assessment is carried out with respect to a set of safety requirements, describing, e.g., the level of tolerance with respect to faults, and the maximum probability that is acceptable for undesired events. Key techniques in this area are *Fault Tree Analysis* (FTA), Failure Modes and Effects Analysis (FMEA), and *Common Cause Analysis* (CCA).

Diagnosability, fault detection and identification analysis Critical systems are designed to be fault tolerant, i.e. they must preserve safety in presence of faults (*fail-safe*), or even be able to operate in presence of faults (*fail-operational*). Fault tolerance typically requires proper architectural design choices (e.g., redundant architectures) and mechanisms to detect (and recover from) faults during operation. Fault Detection and Identification (FDI) analysis has the goal to assess the capabilities of the fault detection and identification mechanisms at design time. Moreover, diagnosability analysis has the goal to evaluate the adequateness of the level of observability (e.g., sensor availability and allocation) for the purpose of diagnosing faults.

The steps of the process are illustrated in detail in Sections 5–7, where they are exemplified on the Triple Modular Generator (TMG) system.

4.2. The XSAP tool

XSAP is built on top of the NUXMV symbolic model checker [CCD+14, nuX19]. NUXMV is an extension of the NUSMV model checker implementing, among others, state-of-the-art techniques for verification of finite- and infinite-state systems, based on SAT (Satisfiability) and SMT (Satisfiability Modulo Theory). XSAP inherits from NUSMV and NUXMV its input language (i.e., SMV).

XSAP relies on an interaction shell similar to the one of NUXMV, which increases the flexibility and possibility of integration within other tools. XSAP provides a *trace viewer* and a *fault tree viewer* to display the generated artifacts graphically. Moreover, it provides *Syntax Directed Editors* (SDEs) for editing models and other input files. XSAP has been developed in C and in C++ for the internal modules, while Python is used for model extension.

XSAP is available for the Linux and MS Windows operating systems. It is distributed under a free license for non-commercial applications or for academic purposes. Custom licensing schemas for industrial usage are possible. The distribution page [xSA19] contains download forms, documentation (including the user manual) and various other resources.

This paper is based on version 1.3.0 of XSAP, released on October 2, 2019.

5. Modeling of the TMG

In this section, we discuss modeling of the TMG, following the XSAP process illustrated in Section 4. We cover the modeling of the nominal behavior, the model extension, the modeling of the controller, and the modeling of the requirements. We also include a general description of the model extension process and the fault libraries, as implemented in XSAP.

5.1. Modeling Nominal Behavior of the TMG

In the rest of the paper we describe in detail the modeling and verification flow of XSAP and we illustrate it on the TMG case study. We discuss the different steps of the process, and the results on the case study. In this section, we start with modeling the nominal behavior of the TMG[2], which is done using the SMV language. We describe the model structure and the modeling of the basic components, while the controller definition is discussed in detail in Section 5.4.

The model of the system is shown in Fig. 6 and it contains two main blocks: SC (System Configuration), containing the system components (generators, circuit breakers and buses) and CN (Controller), implementing the monitoring and control part of the system. For simplicity, we do not show the buses, since their state is fully determined by the state of the other components (we will come back to this point later). For modeling convenience, multiple signals are grouped into arrays of signals when propagated from/to the SC and the CN blocks. The skeleton of the model in SMV, corresponding to Fig. 6, is shown in Listing 1. Blocks are implemented as SMV modules. An SMV module called *main* acts as a container and instantiates the SC and CN modules. Notation 'cmd_Gs[index_G1]' indicates array selection and corresponds to 'cmd_G1' (similarly for the other generators and circuit breakers).

System Configuration

The System Configuration block contains the generators (G1, G2 and G3) and the circuit breakers (GB1, GB2, GB3, BB1, BB2 and BB3). Buses (B1, B2 and B3) are omitted, as previously mentioned.

Generators

Each generator takes as input a command (from the Controller) and an initialization signal. Initialization values are provided as a list of constants in the SC module and localized in one declaration section – this makes it easier to modify the initial state of the system, if needed. Moreover, each generator declares two input events, one failure event (*fev_off*) and one nominal (repair) event (*nev*), representing failures and repairs that may affect a generator[3]. The logic of a generator is implemented in the corresponding module (see Listing 2). A generator has an internal state which is either *on* or *off*. State changes are commanded by the Controller via the input command *cmd*, which can take values *cmd_on*, *cmd_off* or *nop*; in the latter case, the state does not change.

Circuit Breakers

Similarly to generators, circuit breakers take as input a command from the Controller and an initialization signal. Similarly, they declare two failure events (*fev_closed*, *fev_open*) and one nominal (repair) event (*nev*). Each circuit breaker (see Listing 3) has an internal state that may change if commanded by the Controller via the input command *cmd*, which can take values *cmd_closed*, *cmd_open* or *nop*; in the latter case, the state does not change.

Buses

The System configuration block also contains the buses (not shown in Fig. 6, for simplicity). The logic of a bus is displayed in Listing 4. We abstract away the functional behavior of a bus, since the only piece of information which is relevant for our purposes, is whether a bus is powered, and by which input connection (each bus has three of them). Consequently, each bus takes as input three Boolean values (true if and only if the bus is powered by the corresponding input connection). Moreover, a bus has an internal state which may be either *working* or *broken*. The state is initially working, and becomes broken in case the bus is powered simultaneously by two or more connections. If broken, a bus remains broken permanently. A bus is powered if it is working, and it is being powered by exactly one input connection.

Controller and Monitor

The Controller block contains one sub-component MN (Monitor), whose goal is to estimate the state of the system. The Monitor takes as input the commands from the Controller, the events and the initialization signals (for generators

[2] The model is available in the XSAP distribution under examples/fe/triple_modular_generator.

[3] In the nominal model these events are disconnected; they get connected as a result of the model extension phase, which associates them with the fault and repair events generated by the fault models (compare Section 5.2 and Section 5.3).

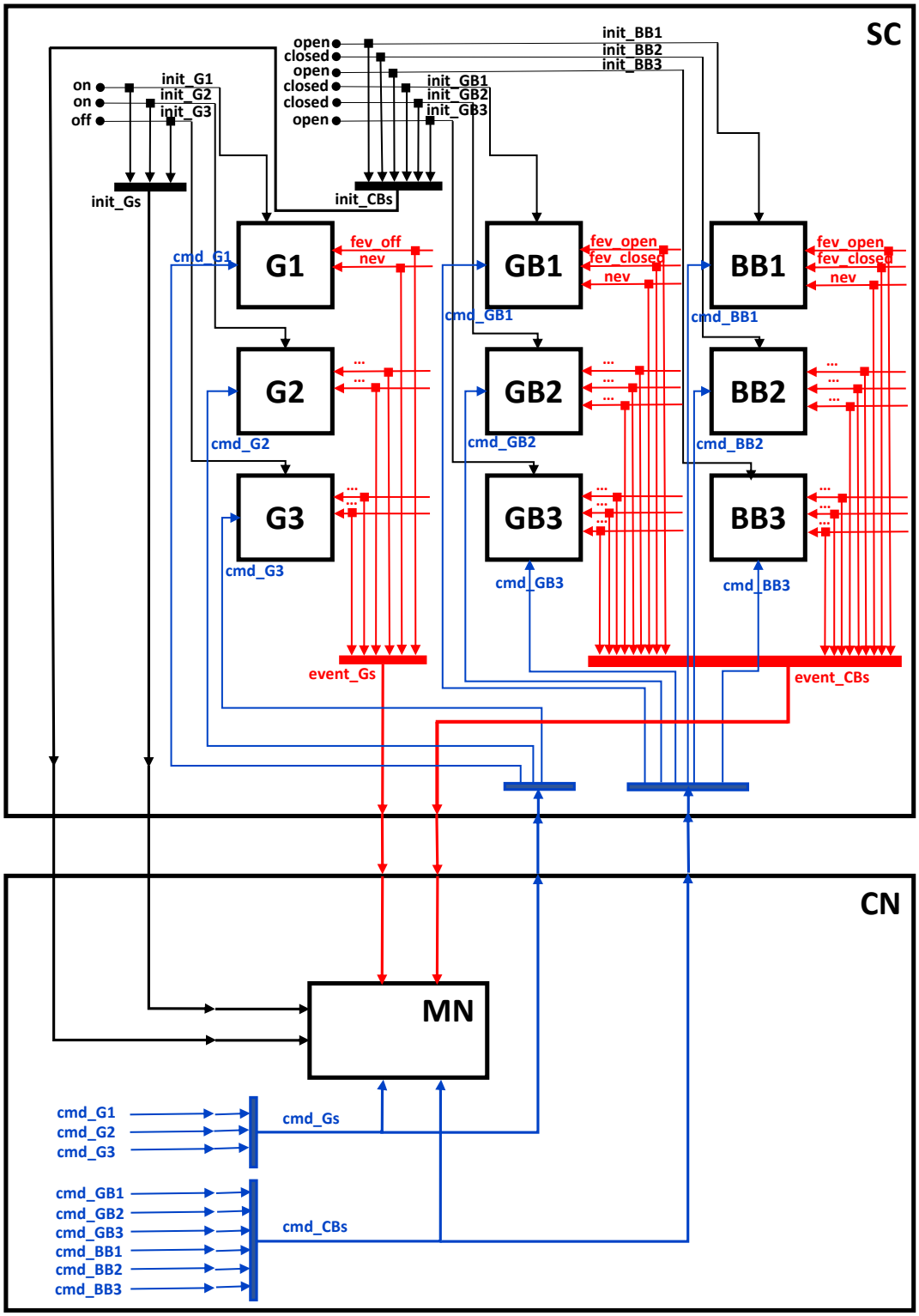

Fig. 6. Architecture of the TMG Model.

```
MODULE main
 VAR
  CN : Controller_Synth(SC.event_Gs, SC.event_CBs, SC.init_Gs, SC.init_CBs);
  SC: System_Configuration(CN.cmd_Gs, CN.cmd_CBs);

MODULE System_Configuration(cmd_Gs, cmd_CBs)
 VAR
  G1 : Generator(cmd_Gs[index_G1], init_G1);
  G2 : Generator(cmd_Gs[index_G2], init_G2);
  G3 : Generator(cmd_Gs[index_G3], init_G3);
  GB1 : Switch(cmd_CBs[index_GB1], init_GB1);
  GB2 : Switch(cmd_CBs[index_GB2], init_GB2);
  GB3 : Switch(cmd_CBs[index_GB3], init_GB3);
  BB1 : Switch(cmd_CBs[index_BB1], init_BB1);
  BB2 : Switch(cmd_CBs[index_BB2], init_BB2);
  BB3 : Switch(cmd_CBs[index_BB3], init_BB3);

MODULE Controller_Synth(event_Gs, event_CBs, init_Gs, init_CBs)
 VAR
  MN : Monitor(event_Gs, event_CBs, cmd_Gs, cmd_CBs, init_Gs, init_CBs);
```

Listing 1. TMG Model Structure in SMV.

```
MODULE Generator(cmd, init_state)
 VAR
  state : {on, off};
 IVAR
  fev_off : boolean;
  nev : boolean;
 DEFINE
  is_on := (state = on);
  is_off := (state = off);
 ASSIGN
 init(state) := init_state;
 next(state) :=
  case
   (cmd = cmd_on) : on;
   (cmd = cmd_off) : off;
   TRUE : state;
  esac;
```

Listing 2. Generator.

```
MODULE Switch(cmd, init_state)
 VAR
  state : {open, closed};
 IVAR
  fev_closed : boolean;
  fev_open : boolean;
  nev : boolean;
 DEFINE
  is_open := (state = open);
  is_closed := (state = closed);
 ASSIGN
  init(state) := init_state;
  next(state) :=
   case
    (cmd = cmd_open) : open;
    (cmd = cmd_closed) : closed;
    TRUE : state;
   esac;
```

Listing 3. Circuit Breaker.

```
MODULE Bus(in1, in2, in3)
 VAR
  state : {working, broken};
 DEFINE
  is_broken := (state = broken);
  is_powered := (state = working) &
          (count((in1), (in2), (in3)) = 1);
 ASSIGN
  init(state) :=
   case
    (count((in1), (in2), (in3)) > 1) : broken;
    TRUE : working;
   esac;
  next(state) :=
   case
    next((count((in1), (in2), (in3)) > 1)) :
                          broken;
    TRUE : state;
   esac;
```

Listing 4. Bus.

and circuit breakers) coming from the System Configuration Block. These inputs are necessary to correctly estimate the state of the individual components and predict the next state. Note that the events represent internal faults (or repairs) of components (compare Footnote 3), hence they may affect the state of the system. The Controller relies on the Monitor to decide the next target state of the system and decide the commands that have to be sent to individual components in order to reach the target state. The Controller and Monitor blocks represent the core of the SMV model of the TMG; their implementation is described later in Section 5.4.

For simplicity, we assume that the Monitor can directly observe faults (and repairs) of components via the corresponding events. This style of modeling can be generalized by adding a set of sensors to the model, and requiring that the monitor can only get information from the sensors, rather than observing the events directly. The extension of the model with observability information is discussed later in Section 7.1.

5.2. Model Extension in XSAP

The process of specifying the set of possible faults and extending the nominal model with such specification is called *model extension*. The result of model extension is the *extended model*, i.e., a model, written in SMV, including the specification of the faulty behaviors. XSAP supports either manual and automatic model extension. In the first scenario,

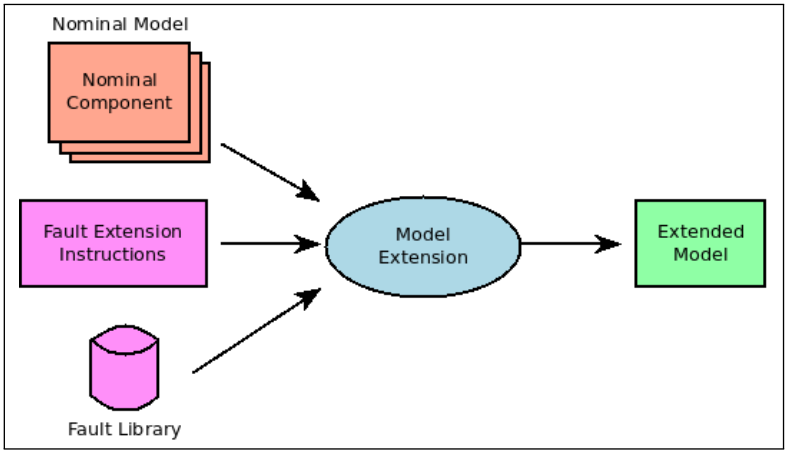

Fig. 7. Automatic model extension.

Name	Parameters	Transitions	
		Entering	During
StuckAtByReference_I	term	next(varout) = next(term)	next(varout) = next(term)
StuckAtByReference_D	term	next(varout) = term	next(varout) = term
StuckAtByValue_I	term	next(varout) = next(term)	next(varout) = varout
StuckAtByValue_D	term	next(varout) = term	next(varout) = varout
StuckAtFixed	—	next(varout) = varout	next(varout) = varout

Fig. 8. Some sample XSAP effects models; column 'Parameters' lists the template parameters other than 'input' and 'varout' that are common to all templates.

the model is written manually, and XSAP provides primitives to declare *fault variables* that control the triggering of the faults. The effects of the faults on the system behavior is encoded manually in the extended model.

In this paper we focus on automatic model extension. In this case, XSAP enables the specification of the faults and their dynamics, by retrieving their definition from a library. Based on the fault specification, the extended model is generated automatically. The effects of the faults on the system behavior are specified in the library and inserted in the nominal model as part of the model extension process. Automatic model extension is illustrated pictorially in Fig. 7.

In the following we illustrate the model extension process more in detail, in particular we describe how to model faults using the fault library, we present the fault extension instructions and discuss how the nominal model is modified with the fault specifications.

5.2.1. Modeling Faults: The Fault Library

The fault library contains *fault templates* for the most common fault definitions. The library contains two different sets of templates (called *effects models* and *dynamics models*) to define *fault effects* and *fault dynamics*. Fault effects specify how the affected part of the system model changes due to the presence of faults, whereas fault dynamics specify how the presence of the fault changes over time (e.g., whether it is a permanent fault, or it can be repaired). The library is extensible and customizable – new templates can be added by the user.

Effects Models Fig. 8 shows a sample list of the XSAP templates for effects models. Specifically, it lists a few variants of the 'StuckAt' template. Fault 'StuckAt' specifies that a given variable become stuck as a result of the fault. Each template, by default, has parameters 'input' and 'varout' (respectively, an input and an output parameter; 'varout' is the name of the variable which is affected by the fault). Moreover, templates may have additional parameters, e.g., all templates in Fig. 8 except the last one, have an additional parameter 'term' (the term at which 'varout' becomes stuck; it may be a constant or a variable).

The two rightmost columns in Fig. 8 specify the fault effect when entering the faulty state ('Entering') and while staying in the faulty state ('During'). There are two main variants of the 'StuckAt' template called *ByReference* and

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

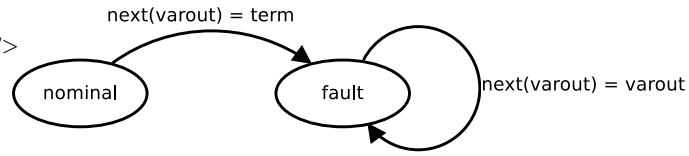

Fig. 9. Effects model for a 'StuckAtByValue_D' fault mode.

ByValue. They have the same effect when entering the faulty state ('varout' gets stuck at 'term'), but they differ in the specification of the 'During' effect: in the call-by-reference case, 'varout' follows 'term', whereas in the call-by-value case, it keeps the 'Entering' value (note that the two cases coincide, if 'term' is a constant). Furthermore, there are two variants (suffix 'I' and suffix 'D' – which stand for 'Instantaneous' and 'Delayed') of the fault template. In the first case, the fault effect propagates instantaneously, whereas in the second case it is applied with a one-step delay (i.e., the next value of 'varout' is computed based on the current value of 'term'). The template 'StuckAtFixed' is a special case where 'varout' gets stuck at its value at the time the fault occurs.

As an example, Fig. 9 (left) shows the definition of the 'StuckAtByValue_D' fault template in the library. The template comes as a file (in xml format) which makes reference to separate files (in SMV format, and also part of the library) containing the definitions of the 'Entering' and 'During' constraints. Semantically, the fault template defines the state machine shown in Fig. 9 (right), where the 'Entering' and 'During' constraints are shown as labels of the transitions between the nominal and faulty state[4].

XSAP provides a few additional fault templates, and several variants thereof, that specify the most common fault effects. For instance, additional templates include 'NonDeterminism' (a numerical variable gets a non-deterministic value within two bounds); 'Conditional' (a variable gets linked to one out of two terms, depending on a Boolean condition evaluated at the time the fault occurs); 'RampDown' (a numerical variable is decremented by a fixed amount at each transition); 'Random' and 'Erroneous' (a variables gets a random value in its domain, in the second case it is required to be different from the current value); 'DeltaOut' (a numerical variable gets displaced w.r.t. its nominal value by a given interval).

Dynamics Models The XSAP templates for dynamics models define the dynamics of the fault. Semantically, they define a state machine describing the transitions from the nominal state and the faulty state. For instance, in the case of a permanent fault, we have a transition from the nominal state to the faulty state, and a self-loop on the faulty state. In case of a transient fault with self-repair, we add a (non-deterministic) transition from the faulty state back to the nominal state. XSAP provides a few templates for the dynamics models, including 'Permanent', 'Transient' and 'SelfFixWithCounter' (a fault may repair itself spontaneously after a given amount of time)[5].

As an example, Fig. 10 (left) shows the definition of the dynamics model for a transient fault with self repair in the library. Similarly to the effects model, the template comes as a file (in xml format) and defines the state machine shown in Fig. 10 (right). The transitions in the state machine define the events that cause a change of state. The *failure* event is a default event that models the occurrence of the fault (transition from the nominal state to the faulty state), whereas the *self_fix* event controls whether the repair non-deterministically takes place (transition from the faulty state back to the nominal state) or whether the fault persists (self-loop on the faulty state; '!' is the negation operator in SMV).

Global Dynamics Model When multiple faults are specified in the same slice, the state machines for the dynamics models of different faults are semantically composed in the following way. The composed state machine has one nominal state which corresponds to sharing the nominal states of the individual state machines. Faulty states of the

[4] In general, the definition of the 'Entering' and 'During' constraints consists of a generic SMV code snippet, making it possible to define arbitrary effects.

[5] The definition of the dynamics model may refer to an additional generic SMV code snippet, which may be used to define arbitrary dynamics. For instance, in the case of the 'SelfFixWithCounter' template, the SMV code defines a counter that is used to trigger the repair transition.

```
<local_dynamics_model name="Transient">
 <templates>
  <template name="self_fix" type="Identifier">
  </template>
 </templates>
 <events>
  <event type="output" name="failure"/>
  <event type="output" name="self_fix"/>
 </events>
 <transitions>
  <transition from="nominal" to="fault">
   <trigger>failure</trigger>
  </transition>
  <transition from="fault" to="fault">
   <guard>!self_fix</guard>
  </transition>
  <transition from="fault" to="nominal">
   <trigger>self_fix</trigger>
  </transition>
 </transitions>
 <raw/>
</local_dynamics_model>
```

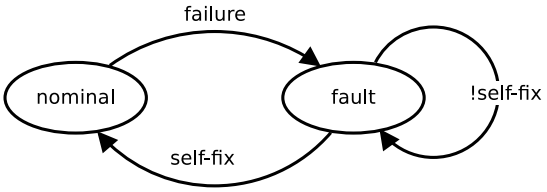

Fig. 10. Dynamics model for a transient fault with self repair.

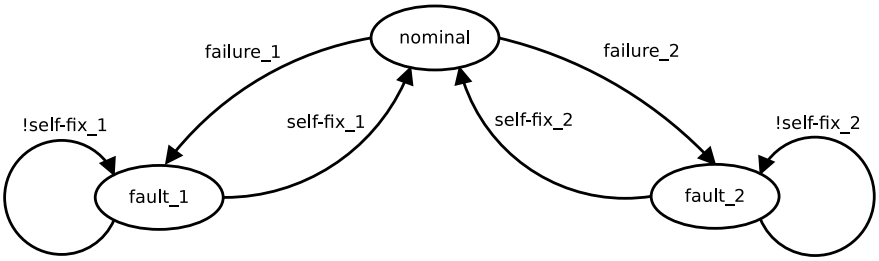

Fig. 11. An example of global dynamics model.

individual state machines are kept separate. For instance, consider two faults 'stuck-at-closed' and 'stuck-at-open' that affect the same component. Each of these two faults semantically defines the state machine shown in Fig. 10 (right). Composing the two states machines semantically yields the state machine shown in Fig. 11, called *global dynamics model*, where the two faulty states and transitions are renamed and duplicated[6]. The rationale of this construction is that different faults for the same component are considered independent (e.g., the 'stuck-at-closed' and 'stuck-at-open' faults are considered independent)[7].

5.2.2. Fault Extension Instructions

The fault extension instructions consist in directives that, for a given nominal model, specify which faults are to be injected into the model, and how the fault templates have to instantiated (i.e., they instantiate the parameters required by the template with actual parameters from the nominal model). Fault extension instructions are specified using an ad-hoc user-readable language called FEI language. The grammar of the language is documented in the XSAP user manual.

5.2.3. Extended Model in SMV

The result of model extension is the *extended model*, i.e. a model that incorporates the definitions of both the nominal and the faulty behaviors, written in SMV. The extended model is produced automatically by XSAP. We conclude this

[6] We remark that the model extension enforces the mutual exclusion between different outgoing events from the same state, e.g. *failure_1* and *failure_2* can not be triggered simultaneously.

[7] For completeness, we mention that in XSAP it is possible to define additional transitions in the composed state machine. As an example, it would be possible to define a transition from *fault_1* to *fault_2* in Fig. 11.

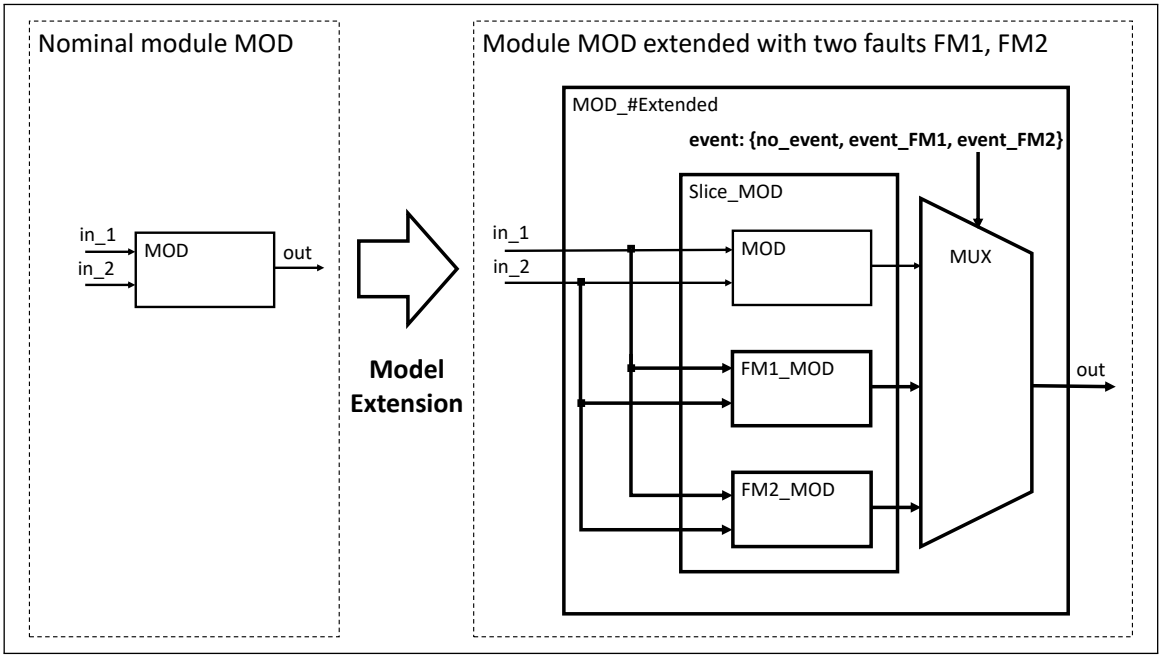

Fig. 12. Model extension at SMV level.

section by describing the structure of the extended model in SMV. The model transformation implemented by XSAP is pictorially illustrated in Fig. 12. Given a nominal module MOD to be extended with the definition of two faults FM1 and FM2, the extended module MOD_#Extended has the structure shown in Fig. 12 (right). It contains a module Slice_MOD corresponding to the fault slice, which in turn contains the definition of the nominal (module MOD) and faulty behaviors defined in the slice (modules FM1_MOD and FM2_MOD). FM1_MOD and FM2_MOD contain the specification of the faulty behaviors (i.e., the code instantiated from the effects model and dynamics model templates). Modules MOD, FM1_MOD and FM2_MOD run in parallel, and the output of the extended module is computed by a multiplexer (MUX), which is controlled by an input event. The input event takes one out of three values: *no_event* (nominal case, the output of MUX is the output of MOD), *event_FM1* (faulty case, the output of MUX is the output of FM1_MOD) and *event_FM2* (faulty case, the output of MUX is the output of FM2_MOD).

5.3. Model Extension of the TMG

We exemplify the model extension on the TMG model. An example FEI specification is given in Listing 5. The extension applies to module types corresponding to the nominal components, namely the generators and circuit breakers[8]. Let us illustrate the extension of the 'Switch' module (line 13), corresponding to the circuit breakers. It is possible to specify sets of faults that affect a common set of variables in the nominal model, such sets are called *slices*[9]. For instance, for the TMG we define a slice called 'Switch_StuckClosed_StuckOpen' (line 15) corresponding to the specification of the 'stuck-at-closed' and 'stuck-at-open' faults of the circuit breakers, which affect variable 'state' (compare Listing 3) of the 'Switch' module (line 16). The slice contains the definition of the two faults, which are declared (line 18) to be instances of the fault templates 'StuckAtByValue_D' (effects model) and 'Transient' (dynamics model), and have an associated fault probability of $1 * 10^{-8}$. Let us describe the instantiation of the 'stuck-at-closed' fault. Lines 19–21 instantiate the formal parameters of the template with actual parameters, in particular the default parameters 'input' and 'varout' are both instantiated with 'state' (the fault reads and modifies the same variable 'state'), whereas the 'term' parameter is instantiated with the constant *closed*. Lines 22-23 link events from the dynamics model template (compare Fig. 10) with input variables in the nominal model. In particular, the *failure* event is linked with event

[8] The language permits a more fine-grained specification that applies to individual instances of modules, which is not illustrated here.
[9] A nominal component can be associated with multiple slices, affecting disjoint sets of variables.

```
1 FAULT EXTENSION FE_SC_TMG
2
3 EXTENSION OF MODULE Generator
4   /-- Description of Fault Slice Gen_StuckOff --/
5   SLICE Gen_StuckOff AFFECTS state WITH
6     /-- Description of fault mode stuckAt_Off --/
7     MODE stuckAt_Off {1.e-7} : Permanent StuckAtByValue_D(
8           data term << off,
9           data input << state,
10          data varout >> state,
11          event failure >> fault_event_stuck_at_off);
12
13 EXTENSION OF MODULE Switch
14   /-- Description of Fault Model for Switch --/
15   SLICE Switch_StuckClosed_StuckOpen
16    AFFECTS state WITH
17     /-- Description of fault mode StuckAt_Closed --/
18     MODE stuckAt_Closed {1.e-8} : Transient StuckAtByValue_D(
19          data term << closed,
20          data input << state,
21          data varout >> state,
22          event failure >> fev_closed,
23          event self_fix >> nev);
24     /-- Description of fault mode StuckAt_Open --/
25     MODE stuckAt_Open {1.e-8} : Transient StuckAtByValue_D(
26          data term << open,
27          data input << state,
28          data varout >> state,
29          event failure >> fev_open,
30          event self_fix >> nev);
```

Listing 5. An example of fault extension instructions.

fev_closed, and the *self_fix* event is linked with event *nev*. We remark that the association of events defined in the dynamics model template with events in the nominal model is optional. In our case, we use this feature to allow the Monitor component to observe failures and repairs (compare discussion at the end of Section 5.1).

The extended model of the TMG is generated automatically by XSAP, based on the nominal model and the FEI specification. The extended SMV model contains extended modules for the basic components (generators and circuit breakers) as described in Fig. 12.

5.4. Requirements-based Modeling of the TMG Controller

In this section we show how to model the controller of the TMG and the corresponding requirements. The goal of the controller is to issue proper commands to the basic components (generators and circuit breakers) in order to preserve the functionality of the system (in a broad sense, as prescribed by its requirements), and minimize the impact of faults. The TMG model comes with a complex and heterogeneous set of requirements (compare Section 3). They focus on different aspects such as safety (e.g., R1), availability (e.g., R2), fault tolerance (e.g., R5), performance/priorities (e.g., R3, R6). Furthermore, some requirements may be potentially conflicting, e.g., it is well known that safety may adversely affect availability (and vice versa).

Synthesizing a controller that is guaranteed to satisfy all the requirements, in all possible system configurations, is not straightforward. Since the set of possible system configurations is finite, in principle it is possible to list all the configurations explicitly, and decide the commands to the issued on a case-by-case basis. However, such approach is impractical and error prone, given the number of possible configurations (nominal and faulty) of system components (considering that generators have 3 possible states and the circuit breakers have 4, this makes $3^3 * 4^6$ configurations which is more that 10^5 configurations). Moreover, this method does not scale to larger systems. In practice, very often simplifying assumptions are made, in order to simplify the synthesis of such systems. For instance, in diagnosis, it is common to limit the number of faults that can be addressed, i.e., in our case we could synthesize the controller under the hypothesis that at most two faults of basic components may occur at any time, thus reducing the overall number of configurations that need to be considered.

In this paper, instead, we take a different approach. Namely, we synthesize the controller using a constraint-based

style of modeling. Specifically, we model directly the requirements listed in Section 3, one-by-one, as properties of the controller, and we impose them as constraints in the nominal model. Note that the requirements are expressed in such a way that they are independent of the history of the system, hence the controller can be specified as a memory-less component. For simplicity, we assume that the controller can react to possible faults in the system instantly, that is, when a fault happens, the system is not allowed to reach an unsafe state (one violating the requirements) since the controller is able to command a change of state that prevents the unsafe situation. This style of modeling can be easily generalized to allow the system to reach an unsafe state and the controller to react with, e.g., a one-step delay to restore a safe state[10].

The result of the modeling is a controller that is guaranteed to satisfy the requirements by construction (since the requirements are part of the model). This approach has two main advantages. First, it is a practical method, which scales for large models. Second, no simplification is needed, in particular, the result can address situations with any number of faults.

Note that the correctness of the synthesized controller w.r.t. the requirements can also be verified as part of functional verification (and safety assessment) of the system. In particular, it is possible to formalize the requirements as properties in temporal logic, check them against the model and prove that they are indeed satisfied (compare Sections 5.5, 6.1 and 6.3). We remark that proving the synthesized controller correct with respect to the requirements also indirectly proves that the original set of requirements is realizable (and the synthesized controller is a correct realization).

We now describe the modeling of the requirements. First, we show the encoding of the Monitor component, which is needed by the Controller to estimate the state of the system. Listing 6 shows an excerpt of the code for the Monitor. In particular, we show the code for circuit breaker BB1 (a similar encoding applies to generators and to the other circuit breakers). The Monitor estimates the mode and the state of BB1. The mode is nominal by default, and can fail open or closed as a result of input failure events; it is also nominal after the occurrence of a repair event; if no events are triggered, the mode does not change. Similarly, the state is computed depending on the occurrence of failure events and on the commands issued by the Controller. For instance, if the circuit breaker is in nominal mode, and the Controller commands it to open, then it opens. If no events are triggered and no commands are issued, the state does not change. Finally, a signal called 'faults_counter' counts the number of faults in the system (i.e., the number of components which are estimated to be in a faulty state).

In the rest of this section, we describe the modeling of each requirement individually. In the following, we refer to code excerpts taken from the Controller component.

R1 "No bus shall be connected to more than one power source at any time".

We illustrate the encoding of R1 for bus B1 in Listing 7 (the encoding for other buses is similar). We enforce B1 to be working (line 18). B1 is working if it is powered by at most one generator (lines 14–16). For each generator G, we list all the possibilities for B1 to be powered (lines 9–11), e.g. 'B1_poweredby_G2' (B1 is powered by G2) holds in two cases (line 10): 'B1_poweredby_G2_L' (B1 is powered by G2 from the left, i.e. through path GB2-BB2-BB3, i.e. if B3 is powered by G2 from the left and BB3 is closed (line 4)); 'B1_poweredby_G2_R' (B1 is powered by G2 from the right, i.e. through path GB2-BB1, i.e. if B2 is powered by G2 from above and BB1 is closed (line 5)). Note that the definitions for different buses are mutually recursive.

R2 "If any power source is on, then all buses will be powered".

R2 is a reliability (availability) requirement that states the level of tolerance of the TMG to faults. Note that it may be violated in presence of a sufficient number of faults. We do not need to model this requirement explicitly as a requirement of the controller, but we can verify it as part of functional verification – we do this in Section 6.1. However, R2 implicitly states that the controller should maximize the availability of the system, that is, the goal of the controller is to ensure that as many buses as possible are powered, insofar this is permitted by faults and while respecting the other requirements. We encode this property of the controller as part of the encoding of requirement R3 below.

R3 "Bus power source priority and source to bus path priority schemes shall be respected at all times (see Fig. 3 and 4)". The encoding of requirement R3 requires to enforce the power source and path priorities for buses B1,

[10] The model with delay is also available in the XSAP distribution under examples/fe/triple_modular_generator. In the generalized model, the system and the controller execute in interleaving, and the execution turn is enforced by a simple scheduler. The implementation of buses needs to be extended to tolerate transient situations when they are exposed to faults, and some requirements have to be modified so that they are evaluated after the execution of the controller.

```
MODULE Monitor(event_Gs, event_CBs, cmd_to_Gs, cmd_to_CBs, init_Gs, init_CBs)
 VAR
  mode_est_BB1 : {ok, ko_open, ko_closed};
  state_est_BB1 : {closed, open};
 DEFINE
  fev_open_CBs := event_CBs[0];
  fev_closed_CBs := event_CBs[1];
  nev_CBs := event_CBs[2];
  index_BB1 := 3;
 ASSIGN
  init(mode_est_BB1) := ok;
  next(mode_est_BB1) :=
   case
    fev_open_CBs[index_BB1] : ko_open;
    fev_closed_CBs[index_BB1] : ko_closed;
    nev_CBs[index_BB1] : ok;
    TRUE : mode_est_BB1;
   esac;
 ASSIGN
  init(state_est_BB1) := init_CBs[index_BB1];
  next(state_est_BB1) :=
   case
    fev_open_CBs[index_BB1] : open;
    fev_closed_CBs[index_BB1] : closed;
 (next(mode_est_BB1) = ok) & (cmd_to_CBs[index_BB1] = cmd_open) : open;
 (next(mode_est_BB1) = ok) & (cmd_to_CBs[index_BB1] = cmd_closed) : closed;
    TRUE : state_est_BB1;
   esac;
-- code for generators and remaining circuit breakers
-- ...
-- faults_counter counts the number of faults in the system
 DEFINE
  faults_counter := count(
   (mode_est_G1 != ok),  (mode_est_G2 != ok),  (mode_est_G3 != ok),
   (mode_est_GB1 != ok), (mode_est_GB2 != ok), (mode_est_GB3 != ok),
   (mode_est_BB1 != ok), (mode_est_BB2 != ok), (mode_est_BB3 != ok));
```

Listing 6. Encoding of the Monitor component.

```
-- Definition of the possible paths to Bus 1 --
DEFINE
 B1_poweredby_G1_U := (next(MN.state_est_G1) = on) & (next(MN.state_est_GB1) = closed);
 B1_poweredby_G2_L := B3_poweredby_G2_L & (next(MN.state_est_BB3) = closed);
 B1_poweredby_G2_R := B2_poweredby_G2_U & (next(MN.state_est_BB1) = closed);
 B1_poweredby_G3_L := B3_poweredby_G3_U & (next(MN.state_est_BB3) = closed);
 B1_poweredby_G3_R := B2_poweredby_G3_R & (next(MN.state_est_BB1) = closed);
DEFINE
 B1_poweredby_G1 := B1_poweredby_G1_U;
 B1_poweredby_G2 := B1_poweredby_G2_R | B1_poweredby_G2_L;
 B1_poweredby_G3 := B1_poweredby_G3_R | B1_poweredby_G3_L;
-- R1: No bus will be connected to more than 1 power source at any time.
DEFINE
 B1_is_working :=
  (B1_poweredby_G1 -> (!B1_poweredby_G2 & !B1_poweredby_G3)) &
  (B1_poweredby_G2 -> (!B1_poweredby_G3));
TRANS
 B1_is_working;
```

Listing 7. Encoding of requirement R1.

```
1 DEFINE
2  stage_1_can_do_B1_G1_U := (next(MN.mode_est_G1) = ok) & (next(MN.mode_est_GB1) in {ok, ko_closed});
3  stage_1_can_do_B1_G2_R := stage_1_can_do_B2_G2_U & (next(MN.mode_est_BB1) in {ok, ko_closed});
4  stage_1_can_do_B1_G2_L := stage_1_can_do_B3_G2_L & (next(MN.mode_est_BB3) in {ok, ko_closed});
5  stage_1_can_do_B1_G3_L := stage_1_can_do_B3_G3_U & (next(MN.mode_est_BB3) in {ok, ko_closed});
6  stage_1_can_do_B1_G3_R := stage_1_can_do_B2_G3_R & (next(MN.mode_est_BB1) in {ok, ko_closed});
7 DEFINE
8  stage_1_can_do_B2_G2_U := (next(MN.mode_est_G2) = ok) & (next(MN.mode_est_GB2) in {ok, ko_closed});
9  stage_1_can_do_B2_G1_L := stage_1_can_do_B1_G1_U & (next(MN.mode_est_BB1) in {ok, ko_closed});
10 stage_1_can_do_B2_G1_R := stage_1_can_do_B3_G1_R & (next(MN.mode_est_BB2) in {ok, ko_closed});
11 stage_1_can_do_B2_G3_R := stage_1_can_do_B3_G3_U & (next(MN.mode_est_BB2) in {ok, ko_closed});
12 stage_1_can_do_B2_G3_L := stage_1_can_do_B1_G3_L & (next(MN.mode_est_BB1) in {ok, ko_closed});
13 DEFINE
14 stage_1_can_do_B3_G2_L := stage_1_can_do_B2_G2_U & (next(MN.mode_est_BB2) in {ok, ko_closed});
15 stage_1_can_do_B3_G2_R := stage_1_can_do_B1_G2_R & (next(MN.mode_est_BB3) in {ok, ko_closed});
16 stage_1_can_do_B3_G1_R := stage_1_can_do_B1_G1_U & (next(MN.mode_est_BB3) in {ok, ko_closed});
17 stage_1_can_do_B3_G1_L := stage_1_can_do_B2_G1_L & (next(MN.mode_est_BB2) in {ok, ko_closed});
18 stage_1_can_do_B3_G3_U := (next(MN.mode_est_G3) = ok) & (next(MN.mode_est_GB3) in {ok, ko_closed});
19 DEFINE
20 stage_1_do_B1 := case
21   stage_1_can_do_B1_G1_U : G1_U;
22   stage_1_can_do_B1_G2_R : G2_R;
23   stage_1_can_do_B1_G2_L : G2_L;
24   stage_1_can_do_B1_G3_L : G3_L;
25   stage_1_can_do_B1_G3_R : G3_R;
26   TRUE : unpowered;
27 esac;
28 DEFINE
29 stage_1_do_B2 := case
30   stage_1_do_B1 in {G2_L, G2_R} : G2_U;
31   stage_1_do_B1 = G3_R : G3_R;
32   stage_1_do_B1 = G1_U & (next(MN.mode_est_BB1) = ko_closed) : G1_L;
33   stage_1_do_B1 = G1_U & (next(MN.mode_est_BB2) = ko_closed) & (next(MN.mode_est_BB3) = ko_closed) : G1_R;
34   stage_1_do_B1 = G3_L & (next(MN.mode_est_BB1) = ko_closed) : G3_L;
35   stage_1_do_B1 = G3_L & (next(MN.mode_est_BB2) = ko_closed) : G3_R;
36   TRUE : unpowered;
37 esac;
38 DEFINE
39 stage_1_do_B3 := case
40   stage_1_do_B1 in {G3_L, G3_R} : G3_U;
41   stage_1_do_B1 = G2_L : G2_L;
42   stage_1_do_B1 = G1_U & (next(MN.mode_est_BB1) = ko_closed) & (next(MN.mode_est_BB2) = ko_closed) : G1_L;
43   stage_1_do_B1 = G1_U & (next(MN.mode_est_BB3) = ko_closed) : G1_R;
44   stage_1_do_B1 = G2_R & (next(MN.mode_est_BB2) = ko_closed) : G2_L;
45   stage_1_do_B1 = G2_R & (next(MN.mode_est_BB3) = ko_closed) : G2_R;
46   TRUE : unpowered;
47 esac;
```

Listing 8. Encoding of requirement R3: stage 1

B2 and B3. The encoding must take into account that priorities are mutually dependent, e.g., enforcing a given path priority for bus B1 may constrain the paths that are possible for bus B2 and B3[11]. Hence, we encode the priorities using a definition in 3 stages (compare Listings 8, 9 and 10), where in stage i, we define the priorities for bus Bi. Note that stage 2 definition depends on stage 1, and stage 3 on stage 1 and 2. Moreover, each stage consists in mutually recursive definitions.

Let us first describe stage 1 definition (Listings 8). Lines 1–18 define the possible paths to power B1, B2 and B3. A path can be used provided that the corresponding generator is working and the required circuit breakers are not failed stuck-at-open. Lines 19–27 define which path has to be used to power B1. The priorities are dictated by the requirements (as per Fig. 3 and 4). Lines 28–37 enforce the consequences of B1 priorities on the possible paths to power B2. Namely, there are two different groups of constraints: lines 30–31 (consequences due to the generator powering B1) and lines 32–36 (consequences due to stuck-at-closed faults). As an example, line 31 states that if B1

[11] Note that the original requirements are under-specified, since they do not prescribe how to resolve such conflicts. Here, we make the choice that choice of the path for bus B1 has priority over the choice for bus B2, which in turn has priority over the choice for bus B3.

```
DEFINE
 stage_2_can_do_B2_G2_U := case
  stage_1_do_B2 = G2_U : TRUE;
  stage_1_do_B2 = unpowered : (next(MN.mode_est_G2) = ok) & (next(MN.mode_est_GB2) in {ok, ko_closed});
  TRUE : FALSE;
 esac;
 stage_2_can_do_B2_G1_L := stage_1_do_B1 = G1_U & (next(MN.mode_est_BB1) in {ok, ko_closed});
 stage_2_can_do_B2_G1_R := stage_2_can_do_B3_G1_R & (next(MN.mode_est_BB2) in {ok, ko_closed});
 stage_2_can_do_B2_G3_R := stage_2_can_do_B3_G3_U & (next(MN.mode_est_BB2) in {ok, ko_closed});
 stage_2_can_do_B2_G3_L := stage_1_do_B1 = G3_L & (next(MN.mode_est_BB1) in {ok, ko_closed});
DEFINE
 stage_2_can_do_B3_G2_L := stage_2_can_do_B2_G2_U & (next(MN.mode_est_BB2) in {ok, ko_closed});
 stage_2_can_do_B3_G2_R := stage_1_do_B1 = G2_R & (next(MN.mode_est_BB3) in {ok, ko_closed});
 stage_2_can_do_B3_G1_R := stage_1_do_B1 = G1_U & (next(MN.mode_est_BB3) in {ok, ko_closed});
 stage_2_can_do_B3_G1_L := stage_2_can_do_B2_G1_L & (next(MN.mode_est_BB2) in {ok, ko_closed});
 stage_2_can_do_B3_G3_U := case
  stage_1_do_B3 = G3_U : TRUE;
  stage_1_do_B3 = unpowered : (next(MN.mode_est_G3) = ok) & (next(MN.mode_est_GB3) in {ok, ko_closed});
  TRUE : FALSE;
 esac;
DEFINE
 stage_2_do_B2 := case
  stage_1_do_B2 != unpowered : stage_1_do_B2;
  stage_2_can_do_B2_G2_U : G2_U;
  stage_2_can_do_B2_G1_L : G1_L;
  stage_2_can_do_B2_G1_R : G1_R;
  stage_2_can_do_B2_G3_R : G3_R;
  stage_2_can_do_B2_G3_L : G3_L;
  TRUE : unpowered;
 esac;
DEFINE
 stage_2_do_B3 := case
  stage_1_do_B3 != unpowered : stage_1_do_B3;
  stage_2_do_B2 = G3_R : G3_U;
  stage_2_do_B2 = G1_R : G1_R;
  stage_2_do_B2 = G2_U & (next(MN.mode_est_BB2) = ko_closed) : G2_L;
  stage_2_do_B2 = G1_L & (next(MN.mode_est_BB2) = ko_closed) : G1_L;
  TRUE : unpowered;
 esac;
```

Listing 9. Encoding of requirement R3: stage 2

```
DEFINE
 stage_3_can_do_B3_G2_L := stage_2_do_B2 = G2_U & (next(MN.mode_est_BB2) in {ok, ko_closed});
 stage_3_can_do_B3_G2_R := stage_1_do_B1 = G2_R & (next(MN.mode_est_BB3) in {ok, ko_closed});
 stage_3_can_do_B3_G1_R := stage_1_do_B1 = G1_U & (next(MN.mode_est_BB3) in {ok, ko_closed});
 stage_3_can_do_B3_G1_L := stage_2_do_B2 = G1_L & (next(MN.mode_est_BB2) in {ok, ko_closed});
 stage_3_can_do_B3_G3_U := case
  stage_2_do_B3 = G3_U : TRUE;
  stage_2_do_B3 = unpowered : (next(MN.mode_est_G3) = ok) & (next(MN.mode_est_GB3) in {ok, ko_closed});
  TRUE : FALSE;
 esac;
DEFINE
 stage_3_do_B3 := case
  stage_2_do_B3 != unpowered : stage_2_do_B3;
  stage_3_can_do_B3_G2_L : G2_L;
  stage_3_can_do_B3_G2_R : G2_R;
  stage_3_can_do_B3_G1_R : G1_R;
  stage_3_can_do_B3_G1_L : G1_L ;
  stage_3_can_do_B3_G3_U : G3_U;
  TRUE : unpowered;
 esac;
```

Listing 10. Encoding of requirement R3: stage 3

```
1  DEFINE
2   cmd_B1_G1_U := (cmd_G1 = cmd_on) & (cmd_GB1 = cmd_closed);
3   cmd_B1_G2_R := cmd_B2_G2_U & (cmd_BB1 = cmd_closed);
4   cmd_B1_G2_L := cmd_B3_G2_L & (cmd_BB3 = cmd_closed);
5   cmd_B1_G3_L := cmd_B3_G3_U & (cmd_BB3 = cmd_closed);
6   cmd_B1_G3_R := cmd_B2_G3_R & (cmd_BB1 = cmd_closed);
7
8  DEFINE
9   cmd_B2_G2_U := (cmd_G2 = cmd_on) & (cmd_GB2 = cmd_closed);
10  cmd_B2_G1_L := cmd_B1_G1_U & (cmd_BB1 = cmd_closed);
11  cmd_B2_G1_R := cmd_B3_G1_R & (cmd_BB2 = cmd_closed);
12  cmd_B2_G3_R := cmd_B3_G3_U & (cmd_BB2 = cmd_closed);
13  cmd_B2_G3_L := cmd_B1_G3_L & (cmd_BB1 = cmd_closed);
14
15 DEFINE
16  cmd_B3_G2_L := cmd_B2_G2_U & (cmd_BB2 = cmd_closed);
17  cmd_B3_G2_R := cmd_B1_G2_R & (cmd_BB3 = cmd_closed);
18  cmd_B3_G1_R := cmd_B1_G1_U & (cmd_BB3 = cmd_closed);
19  cmd_B3_G1_L := cmd_B2_G1_L & (cmd_BB2 = cmd_closed);
20  cmd_B3_G3_U := (cmd_G3 = cmd_on) & (cmd_GB3 = cmd_closed);
21 -- B1 preferences
22 TRANS
23  case
24   stage_1_do_B1 = G1_U : cmd_B1_G1_U;
25   stage_1_do_B1 = G2_R : cmd_B1_G2_R;
26   stage_1_do_B1 = G2_L : cmd_B1_G2_L;
27   stage_1_do_B1 = G3_L : cmd_B1_G3_L;
28   stage_1_do_B1 = G3_R : cmd_B1_G3_R;
29   TRUE : TRUE;
30  esac;
31 -- B2 preferences
32 TRANS
33  case
34   stage_2_do_B2 = G2_U : cmd_B2_G2_U;
35   stage_2_do_B2 = G1_L : cmd_B2_G1_L;
36   stage_2_do_B2 = G1_R : cmd_B2_G1_R;
37   stage_2_do_B2 = G3_R : cmd_B2_G3_R;
38   stage_2_do_B2 = G3_L : cmd_B2_G3_L;
39   TRUE : TRUE;
40  esac;
41 -- B3 preferences
42 TRANS
43  case
44   stage_3_do_B3 = G2_L : cmd_B3_G2_L;
45   stage_3_do_B3 = G2_R : cmd_B3_G2_R;
46   stage_3_do_B3 = G1_R : cmd_B3_G1_R;
47   stage_3_do_B3 = G1_L : cmd_B3_G1_L;
48   stage_3_do_B3 = G3_U : cmd_B3_G3_U;
49   TRUE : TRUE;
50  esac;
```

Listing 11. Encoding of requirement R3: enforcing priorities

is powered by G3 from the right, then B2 must also be powered by G3 from the right, and line 35 states that if B1 is powered by G3 from the left and BB2 is failed stuck-at-closed, then B2 must be powered by G3 from the right. Similarly, lines 38–47 enforce the consequences of B1 priorities on the possible paths to power B3.

In stage 2 definition (Listings 9) we define the possible paths to power B2 and B3. The definition for the possible paths is similar to stage 1, however it has to take into account the decisions for B1 taken in stage 1. For instance, in line 7, powering B2 using G1 from the left is possible only if stage 1 forced B1 to be powered by G1 from above. Lines 21–30 define the priorities for B2. If a priority for B2 has already been decided in stage 1, then it is propagated here, otherwise it is decided based on the priority requirements. Lines 31–39 enforce the consequences of B2 priorities on the possible paths to power B3.

Stage 3 (Listings 10) follows the same pattern as stage 2.

Finally, Listing 11 describes how we enforce the priorities defined so far on the model. Lines 1–20 define the

```
1 VAR
2  B1 : Bus(B1_poweredby_G1, B1_poweredby_G2, B1_poweredby_G3);
3 DEFINE
4  B1_poweredby_G1_U := (G1.is_on) & (GB1.is_closed);
5  B1_poweredby_G2_L := B3_poweredby_G2_L & (BB3.is_closed);
6  B1_poweredby_G2_R := B2_poweredby_G2_U & (BB1.is_closed);
7  B1_poweredby_G3_L := B3_poweredby_G3_U & (BB3.is_closed);
8  B1_poweredby_G3_R := B2_poweredby_G3_R & (BB1.is_closed);
9 DEFINE
10  B1_poweredby_G1 := B1_poweredby_G1_U;
11  B1_poweredby_G2 := B1_poweredby_G2_R | B1_poweredby_G2_L;
12  B1_poweredby_G3 := B1_poweredby_G3_R | B1_poweredby_G3_L;
```

Listing 12. Encoding of requirement R4.

commands that realize a given source-path power option. For instance (line 2) powering B1 using G1 from above requires commanding G1 to switch on, and GB1 to close. Lines 22-50 translate the priorities for B1, B2 and B3 in the corresponding commands for the controller (note that conditions in a *case* statement are evaluated in order).

R4 "If no power source is on, then all buses will be unpowered". We define the property of a bus being powered with the following declaration in the Bus module: DEFINE is_powered := (state = working) & (count((in1), (in2), (in3)) = 1), that is, a bus is powered if it is working and exactly one out of the three inputs in1, in2, in3 is true (the bus is powered by exactly one generator). We illustrate the encoding of R4 for bus B1 in Listing 12. Bus B1 is instantiated as in line 2. The input parameters are defined in the constraints below (lines 4–8 and 10–12), which follow the same pattern as for requirements R1 and R3. It is easy to see that these constraints state that B1 is powered if it is electrically connected to one generator that is switched on. As a consequence, if no generator is switched on, then B1 is unpowered, and similarly for the other buses.

R5 "Any single/dual component failure shall not cause other system requirements to be violated."

R2 is a reliability (fault tolerance) requirement that states that the system is resilient to single and double faults. We do not need to model this requirement explicitly as a requirement of the controller; we will formalize and verify it as part of functional verification in Section 6.1. Note that, as for requirement R2, it is important to enforce that the controller maximizes the availability of the system (see requirement R3).

R6 "Never more than two generators are on, unless required in case of failures".

Requirement R6 states that at most two generators should be switched on at any time, except in presence of faults. This requirements is a consequence of requirements R1, R3 and of the maximum availability property of the controller. In Section 6.1 we will formalize this requirement and prove more precisely under which circumstances (number of faults) it is necessary to switch on all three generators.

5.5. Modeling the Properties of the TMG

In this section we discuss the modeling of the properties of the TMG. We use these properties in Section 6 to carry out functional verification and safety assessment of the TMG.

We show a sample list of properties in Listing 13. Some properties are directly derived from the requirements of the TMG described in Section 3, whereas some others are used as sanity checks or to prove further characteristics of the system. Properties are specified in the main module. They are expressed as invariants or properties written in temporal logic (CTL or LTL). Property 'R1_true' states that buses never break. This is a consequence of requirements R1 that we imposed on the controller, namely that buses are never connected to more than one power source at any time. Property 'R2_false' states that all the buses are always powered. We can expect this property to be falsified in case of (multiple) faults. We interpret this property in view of requirement R5 (fault tolerance to single and double faults) and amend it into 'R2_true' (we add the assumption that the number of faults is less than 3 – note that 'faults_counter' is defined in the Monitor module, compare Listing 6). 'R4_true' states that all buses are unpowered, if all generators are switched off. 'R6_true' states that, in case there are no faults, at most two generators are switched on at any time.

The remaining properties state various characteristics of the TMG. Property 'Monitor_State_Consistency_true' states that the state estimation of the monitor is always correct (i.e., the estimated state matches the real state). Property 'system_in_init_configuration_until_fault_true' states that the system always stays in the initial configuration, unless a

```
1 -- R1: No bus will be connected to more than 1 power source at any time.
2 INVARSPEC NAME R1_true := !(SC.B1.is_broken | SC.B2.is_broken | SC.B3.is_broken);
3
4 -- R2: all buses are always powered
5 -- R5: Any single/dual component failure shall not cause other system requirements to be violated.
6 INVARSPEC NAME R2_false := (SC.B1.is_powered & SC.B2.is_powered & SC.B3.is_powered);
7 INVARSPEC NAME R2_true := ((CN.MN.faults_counter < 3) -> (SC.B1.is_powered & SC.B2.is_powered & SC.B3.is_powered));
8
9 -- R4: If no power source is on, then all buses will be unpowered
10 INVARSPEC NAME R4_true :=
11  ((SC.G1.is_off & SC.G2.is_off & SC.G3.is_off) -> (!SC.B1.is_powered & !SC.B2.is_powered & !SC.B3.is_powered));
12
13 -- R6: Never more than two generators on unless required in case of failure
14 INVARSPEC NAME R6_true :=
15  (CN.MN.faults_counter = 0) -> (SC.G1.is_off | SC.G2.is_off | SC.G3.is_off);
16
17 -- The monitor correctly knows the state of the system at anytime
18 INVARSPEC NAME Monitor_State_Consistency_true :=
19  (SC.G1.state  = CN.MN.state_est_G1)  & (SC.G2.state  = CN.MN.state_est_G2)  & (SC.G3.state = CN.MN.state_est_G3) &
20  (SC.GB1.state = CN.MN.state_est_GB1) & (SC.GB2.state = CN.MN.state_est_GB2) & (SC.GB3.state = CN.MN.state_est_GB3) &
21  (SC.BB1.state = CN.MN.state_est_BB1) & (SC.BB2.state = CN.MN.state_est_BB2) & (SC.BB3.state = CN.MN.state_est_BB3);
22
23 -- If there are no faults, the system stays in the initial configuration
24 DEFINE init_conf :=
25  SC.G1.is_on & SC.G2.is_on & SC.G3.is_off &
26  SC.GB1.is_closed & SC.GB2.is_closed & SC.GB3.is_open &
27  SC.BB1.is_open & SC.BB2.is_closed & SC.BB3.is_open;
28 LTLSPEC NAME system_in_init_configuration_until_fault_true := (init_conf U (CN.MN.faults_counter > 0)) | G init_conf;
29
30 -- There exists a path in which there is no fault
31 CTLSPEC NAME system_can_always_work_true := EG (CN.MN.faults_counter = 0);
32
33 -- From each nominal state it is always possible to end up in a faulty state
34 CTLSPEC NAME system_can_always_fail_true :=
35  AG(CN.MN.faults_counter = 0 -> EX (CN.MN.faults_counter >0));
36 CTLSPEC NAME G1_can_always_fail_true := AG( CN.MN.mode_est_G1 = ok -> EX (CN.MN.mode_est_G1 = ko));
37 CTLSPEC NAME BB1_can_always_fail_true :=
38  AG(CN.MN.mode_est_BB1 = ok -> EX (CN.MN.mode_est_BB1 = ko_open) & EX (CN.MN.mode_est_BB1 = ko_closed));
```

Listing 13. Properties of the TMG.

fault occurs. Finally, we use some CTL properties to express 'sanity checks', namely to assess that the behavior of the model matches the designer's intent. In particular, property 'system_can_always_work_true' states that there exists an execution where no faults occur. 'system_can_always_fail_true' states that it is always possible for a fault to occur in a nominal state. Finally, properties 'G1_can_always_fail_true' and 'BB1_can_always_fail_true' state that there exists an execution where G1 (respectively, BB1) fails; similar properties can be formulated for the remaining generators and circuit breakers.

6. Functional Verification and Safety Assessment of the TMG

In this section we discuss functional verification and safety assessment. We recall the main definitions we need for safety assessment, in particular the definitions of fault trees and (minimal) cut sets. We also discuss Failure Modes and Effects Analysis (FMEA), Common Cause Analysis (CCA) and latent faults. Then, we apply and illustrate the outcome of the analyses on the TMG model.

6.1. Functional Verification of the TMG

We now illustrate the formal verification of the TMG, against the properties identified in Section 5.5. The properties can be verified using the property verification primitives in XSAP, which are inherited from the NUXMV model checker.

All properties are verified to hold, with the exception of 'R2_false' and 'system_in_init_configuration_until_fault_true'.

```
TRANS
  ((next(MN.state_est_GB1) = open) −> (cmd_G1 = cmd_off)) &
  ((next(MN.state_est_GB2) = open) −> (cmd_G2 = cmd_off)) &
  ((next(MN.state_est_GB3) = open) −> (cmd_G3 = cmd_off));
TRANS
  ((next(MN.mode_est_GB1) = ok & next(MN.state_est_GB1) = closed) −> (cmd_G1 = cmd_on)) &
  ((next(MN.mode_est_GB2) = ok & next(MN.state_est_GB2) = closed) −> (cmd_G2 = cmd_on)) &
  ((next(MN.mode_est_GB3) = ok & next(MN.state_est_GB3) = closed) −> (cmd_G3 = cmd_on));
```

Listing 14. Constraints to resolve non-determinism in the Controller.

```
000 :EG CN.MN.faults_counter = 0
  [CTL        True        N/A   system_can_always_work_true]
001 :AG (CN.MN.faults_counter = 0 −> EX CN.MN.faults_counter > 0)
  [CTL        True        N/A   system_can_always_fail_true]
002 :AG (CN.MN.mode_est_G1 = ok −> EX CN.MN.mode_est_G1 = ko)
  [CTL        True        N/A   G1_can_always_fail_true]
<...>
008 :AG (CN.MN.mode_est_BB1 = ok −> (EX CN.MN.mode_est_BB1 = ko_open & EX CN.MN.mode_est_BB1 = ko_closed))
  [CTL        True        N/A   BB1_can_always_fail_true]
<...>
011 :((init_conf U CN.MN.faults_counter > 0) | G init_conf)
  [LTL        True        N/A   system_in_init_configuration_until_fault_true]
012 :!((SC.B1.is_broken | SC.B2.is_broken) | SC.B3.is_broken)
  [Invar      True        N/A   R1_true]
013 :((SC.B1.is_powered & SC.B2.is_powered) & SC.B3.is_powered)
  [Invar      False       1     R2_false]
014 :(CN.MN.faults_counter < 3 −> ((SC.B1.is_powered & SC.B2.is_powered) & SC.B3.is_powered))
  [Invar      True        N/A   R2_true]
015 :(((SC.G1.is_off & SC.G2.is_off) & SC.G3.is_off) −> ((!SC.B1.is_powered & !SC.B2.is_powered) & !SC.B3.is_powered))
  [Invar      True        N/A   R4_true]
016 :(CN.MN.faults_counter = 0 −> ((SC.G1.is_off | SC.G2.is_off) | SC.G3.is_off))
  [Invar      True        N/A   R6_true]
017 :(((((((SC.G1.state = CN.MN.state_G1 & SC.G2.state = CN.MN.state_G2) & SC.G3.state = CN.MN.state_G3) &
        SC.GB1.state = CN.MN.state_GB1) & SC.GB2.state = CN.MN.state_GB2) & SC.GB3.state = CN.MN.state_GB3) &
        SC.BB1.state = CN.MN.state_BB1) & SC.BB2.state = CN.MN.state_BB2) & SC.BB3.state = CN.MN.state_BB3)
  [Invar      True        N/A   Monitor_State_Consistency_true]
```

Listing 15. Outcome of property verification using XSAP.

'R2_false' (all buses are always powered) is clearly proved false, since it may not hold in presence of (multiple) faults. 'R2_true' holds instead, proving that, under the hypothesis that there are at most two faults, all buses are always powered.

Property 'system_in_init_configuration_until_fault_true' states that the system always stays in the initial configuration, unless a fault occurs. An analysis of the counterexample generated by XSAP shows that this property does not hold, since there is some non-determinism in the controller specification. For instance, considering the pair G1-GB1, there are two ways to disable it (i.e., achieve a situation where no current flows out of GB1): switch off G1 (regardless of the state of GB1) or open GB1 (regardless of the state of G1). To avoid this non-determinism, we add the constraints shown in Listing 14. Namely, we state that when GB1 is open, G1 must be switched off, and when GB1 is ok and closed, G1 must be switched on. Similarly for the pairs G2-GB2 and G3-GB3.

With this modification, property 'system_in_init_configuration_until_fault_true' holds. Listing 15 shows the XSAP property database, summarizing the outcome of property verification.

We remark that property 'R6_true' can be generalized to single faults. It is easy to prove that even in presence of single faults, the controller needs to switch on at most two generators. The property cannot be generalized to double faults, e.g., in case both BB2 and BB3 fail, the path priorities force the controller to switch on all generators.

6.2. Safety Assessment in XSAP

Safety assessment has the goal to assess the dependability characteristics and fault tolerance of a given system. Typical artifacts produced by safety assessment are fault trees and FMEA tables [VSD+02, SAE96]. In many application domains, e.g. aeronautics, such activities are mandatory to obtain system certification.

6.2.1. Fault Tree Analysis

A Fault Tree (FT) is a graphical representation of the sets of possible causes of a given (undesired) event called *Top Level Event (TLE)*. The TLE acts as the root of the tree and is linked by means of logical gates (AND, OR) to the basic events (faults). The minimal combinations of faults explaining the TLE are called *Minimal Cut Sets (MCSs)*. A fault tree is a qualitative artifact that is typically evaluated quantitatively. In particular, the probability of the TLE and the intermediate nodes of the tree are computed depending on the probabilities of the basic events, which are assumed to be independent[12].

We give the following definitions. We assume that a system is given, represented as an STS. A cut set is formally defined as follows [BCT07].

Definition 6.1 (Cut set). Let $S = \langle V, V_o, W, W_o, F, I, T\rangle$ be an STS, $FC \subseteq F$ a set of faults (*fault configuration*) , and TLE a formula over V (top level event). We say that FC is a cut set of TLE, written $cs(FC, TLE)$ if there exists a trace $\pi = s_0, i_1, s_1, i_2, s_2, \ldots, i_k, s_k$ of S such that $s_k \models TLE$ and $\forall f \in F.\ f \in FC \iff \exists i \in \{0, \ldots, k\}.\ (s_i \models f)$.

Intuitively, a cut set is a set of faults (called a fault configuration), each of them being active at some point along a trace witnessing the occurrence of the top level event. The cardinality of the cut set is called *order*.

Minimal Cut Sets (MCSs) are minimal fault configurations satisfying Definition 6.1. In safety analysis, it is important to identify the minimal cut sets, since they represent simpler explanations for the top level event and, under the assumption of independent faults, they have higher probabilities. MCSs are defined as follows.

Definition 6.2 (Minimal Cut Sets). Let $S = \langle V, V_o, W, W_o, F, I, T\rangle$ be an STS, $FConf = 2^F$ the set of all fault configurations, and TLE a top level event. We define the set of cut sets and minimal cut sets of TLE as follows:

$$\begin{array}{lcl} \mathrm{CS}(TLE) & = & \{FC \in FConf \mid cs(FC, TLE)\} \\ \mathrm{MCS}(TLE) & = & \{cs \in \mathrm{CS}(TLE) \mid \forall cs' \in \mathrm{CS}(TLE)\ (cs' \subseteq cs \Rightarrow cs' = cs)\} \end{array}$$

The definition of MCSs is inherently based on the assumption that fault configurations are *monotonic*, i.e. activating additional faults cannot prevent the occurrence of the top level event. This assumption is commonly used in standard practice, since it leads to a conservative over-approximation of the probability of the TLE (i.e., system unreliability). Typically, the over-approximation is acceptable, since the probability that a fault does *not* occur (calculated as 1 minus the probability of the fault occurring) is usually "close to 1".

6.2.2. Failure Modes and Effects Analysis

FMEA tables are a tabular representation of the causality relationships between (sets of) faults and a list of properties (undesired events). FMEA takes as input a set of fault configurations and a set of top level events, and it produces a mapping (tabular representation) between elements in the two sets. An entry in the table means that a given fault configuration is a possible explanation for the corresponding top level event.

Definition 6.3 (FMEA Tables). Let $S = \langle V, V_o, W, W_o, F, I, T\rangle$ be an STS, $F = \{FC_1, \ldots FC_l\} \subseteq 2^F$ a set of fault configurations, and $T = \{TLE_1, \ldots, TLE_m\}$ a set of top level events. An FMEA table for F and T, denoted $\mathrm{FMEA}(F, T)$, is the set of pairs $\{\langle FC_i, TLE_j\rangle \mid FC_i \in F \wedge TLE_j \in T \wedge cs(FC_i, TLE_j)\}$

Usually, FMEA tables are generated for all possible fault configurations up to a maximum cardinality (called *order*), e.g., an FMEA table of order 2 considers all fault configurations with at most 2 faults. Often, only single faults are considered, i.e. FMEA tables of order 1 are generated.

6.2.3. Common Cause Analysis

Common Cause Analysis (CCA) is an important step of safety assessment. Its purpose is to evaluate the consequences of events that may break the hypothesis of independence of faults. For instance, two components that are physically located close to each other, may break simultaneously as a result of a common initiating event (the *common cause*),

[12] Depending on the domain, the fault probability may be expressed in different ways, e.g., as probability per unit of time (called *failure rate*) or probability per mission, with reference to a default mission duration. In the context of this paper we need not commit to a specific choice. For the TMG model we can assume that the fault probabilities are expressed as failure rates, without loss of generality.

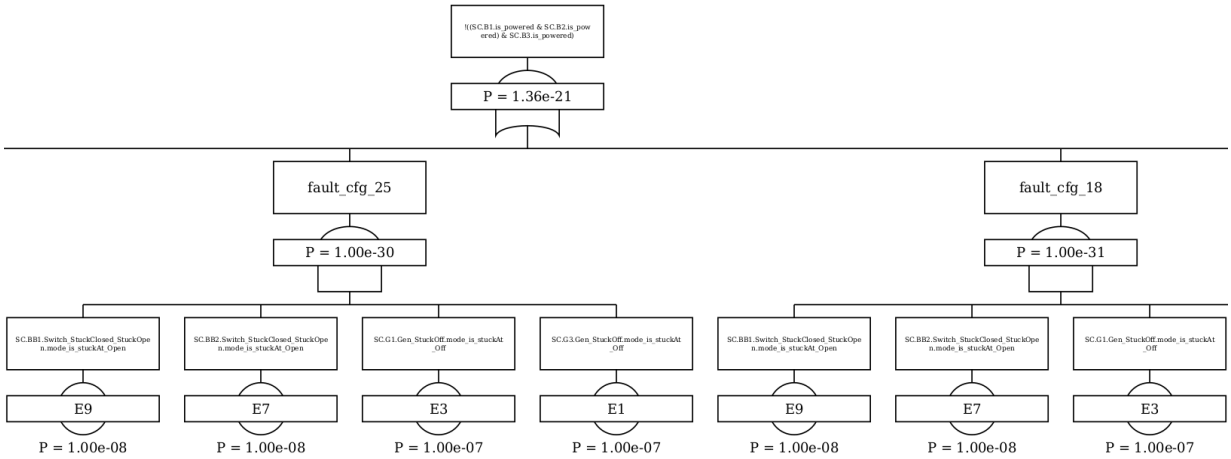

Fig. 13. A fault tree for the TMG (excerpt).

e.g., a fire. In other situations, components may break as a result of a logical dependence, e.g. two components that are commanded by the same software, may break due to a wrong control strategy.

Common causes must be investigated since they invalidate the fault independence assumption, thus leading to higher probability of failure (namely, the probability of a fault combination may be much higher than the product of the individual probabilities). As part of safety assessment, potential common causes are identified, and their consequences on system reliability investigated.

6.2.4. Latent Faults

A latent fault is a fault which is present but has not been detected at the time the mission starts (e.g., at the beginning of a flight for an aircraft, or at the time a system is put into operation). For instance, a latent fault may be associated with a component which is not in use, or may be such that its effects are masked due to, e.g., the operational conditions or mission phase, or due to fault tolerance mechanisms implemented in the system. A latent fault may have an impact on safety, since it decreases the reliability of the system.

XSAP enables the specification of faults that may be failed latent. For such faults, the *latent probability* (probability that the fault is already present before the mission starts) is specified in the fault extension instructions in addition to the standard probability (probability that the fault randomly occurs during the mission).

6.3. Safety Assessment of the TMG

In this section we illustrate the safety assessment process on the TMG model. In particular, we use XSAP to generate fault trees and FMEA tables, and we evaluate the dependability characteristics of the TMG.

XSAP can automatically construct a fault tree, taking as input the extended model of the TMG, the fault specification, and a formula corresponding to the top level event. The fault tree is a logical characterization of the minimal cut sets, according to Definition 6.2. We generate the fault tree for the top level event corresponding to the negation of invariant 'R2_false' (i.e., stating that at least one bus is not powered). Fig. 13 shows a portion of the generated fault tree, where two minimal cut sets (of order 4 and 3, respectively) are visible. Each cut set represents a fault configuration that causes the top level event (a possible cause for a bus not being powered). The basic events are decorated with the probabilities specified in the fault extension instructions (compare Listing 5). The probabilities of the minimal cut sets and of the top level event are computed based on the probabilities of the basic events. Fig. 14 lists all the minimal cut sets of order 3, along with some statistics, generated by XSAP: the fault tree has 14 minimal cut sets of order 3 and 12 minimal cut sets of order 4, for a total of 26 minimal cut sets. An example of minimal cut set of order 4, corresponding to the one displayed in Fig. 13, is 'G1 stuck off, G3 stuck off, BB1 stuck open, BB2 stuck open'.

Considering the same top level event (the negation of 'R2_false'), the FMEA tables of order up to 2 are empty. In fact, there are no cut sets or order up to 2. The FMEA table of order 3 contains the minimal cut sets listed in Fig. 14,

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

NR	MCS		
1	GB1 stuck open,	GB2 stuck open,	GB3 stuck open
2	GB2 stuck open,	BB1 stuck open,	BB2 stuck open
3	GB1 stuck open,	BB1 stuck open,	BB3 stuck open
4	GB3 stuck open,	BB2 stuck open,	BB3 stuck open
5	G1 stuck off ,	GB2 stuck open,	GB3 stuck open
6	G1 stuck off ,	BB1 stuck open,	BB3 stuck open
7	G2 stuck off ,	GB1 stuck open,	GB3 stuck open
8	G2 stuck off ,	BB1 stuck open,	BB2 stuck open
9	G1 stuck off ,	G2 stuck off ,	GB3 stuck open
10	G3 stuck off ,	GB1 stuck open,	GB2 stuck open
11	G3 stuck off ,	BB2 stuck open,	BB3 stuck open
12	G1 stuck off ,	G3 stuck off ,	GB2 stuck open
13	G2 stuck off ,	G3 stuck off ,	GB1 stuck open
14	G1 stuck off ,	G2 stuck off ,	G3 stuck off

Fig. 14. Statistics and minimal cut sets or order 3 for the fault tree in Fig. 13, with TLE 'R2_false' (at least one bus is not powered).

```
def fault_probability(
  PROB__hist_var_13_SC_GB2_Switch_StuckClosed_StuckOpen_mode_is_stuckAt_Open_TRUE,
  PROB__hist_var_15_SC_GB1_Switch_StuckClosed_StuckOpen_mode_is_stuckAt_Open_TRUE,
  PROB__hist_var_5_SC_BB3_Switch_StuckClosed_StuckOpen_mode_is_stuckAt_Open_TRUE,
  PROB__hist_var_7_SC_BB2_Switch_StuckClosed_StuckOpen_mode_is_stuckAt_Open_TRUE,
  PROB__hist_var_2_SC_G2_Gen_StuckOff_mode_is_stuckAt_Off_TRUE,
  PROB__hist_var_3_SC_G1_Gen_StuckOff_mode_is_stuckAt_Off_TRUE,
  PROB__hist_var_11_SC_GB3_Switch_StuckClosed_StuckOpen_mode_is_stuckAt_Open_TRUE,
  PROB__hist_var_9_SC_BB1_Switch_StuckClosed_StuckOpen_mode_is_stuckAt_Open_TRUE,
  PROB__hist_var_1_SC_G3_Gen_StuckOff_mode_is_stuckAt_Off_TRUE):
    EXPR_1 = PROB__hist_var_13_SC_GB2_Switch_StuckClosed_StuckOpen_mode_is_stuckAt_Open_TRUE
    EXPR_2 = PROB__hist_var_15_SC_GB1_Switch_StuckClosed_StuckOpen_mode_is_stuckAt_Open_TRUE
    EXPR_3 = PROB__hist_var_5_SC_BB3_Switch_StuckClosed_StuckOpen_mode_is_stuckAt_Open_TRUE
    EXPR_4 = PROB__hist_var_7_SC_BB2_Switch_StuckClosed_StuckOpen_mode_is_stuckAt_Open_TRUE
    EXPR_5 = 1
    EXPR_6 = EXPR_5 - EXPR_4
    EXPR_7 = PROB__hist_var_11_SC_GB3_Switch_StuckClosed_StuckOpen_mode_is_stuckAt_Open_TRUE
    EXPR_8 = EXPR_5 - EXPR_7
    EXPR_9 = PROB__hist_var_9_SC_BB1_Switch_StuckClosed_StuckOpen_mode_is_stuckAt_Open_TRUE
    EXPR_10 = EXPR_5 - EXPR_9
    <...>
    EXPR_100 = EXPR_75 + EXPR_99
    EXPR_101 = EXPR_55 * EXPR_100
    EXPR_102 = EXPR_54 + EXPR_101
    prob = EXPR_102
    return prob
if __name__ == '__main__':
    p = fault_probability()
    print("Default probability of TLE is: %e\n" % p)
```

Fig. 15. Symbolic probability function for the fault tree in Fig. 13 (excerpt).

whereas the FMEA or order 4 contains, in addition to the minimal cut sets of order 3 and 4, all (non minimal) fault configurations or cardinality 4 that are supersets of a minimal cut set of order 3.

6.3.1. Symbolic Probability Computation

Along with the computation of the probability for the top level event, XSAP produces a symbolic function called *unreliability function* that can be used to compute the probability of the top level event, for generic values of the probabilities of the basic events (that is, the probabilities of the basic events are parameters of this function). Fig. 15 shows the symbolic probability function, in python syntax, for the fault tree in Fig. 13.

This feature can be used to re-evaluate the probability of the top level event for different values of the probabilities

```
1 FAULT EXTENSION FE_SC_TMG
2 /-- ... --/
3 COMMON CAUSES
4   CAUSE CC1 {1.5e-8}
5     MODULE Generator
6       FOR INSTANCES SC.G[12]
7       MODE Gen_StuckOff.stuckAt_Off WITHIN 1 .. 2;
```

Listing 16. Fault extension instructions for a common cause.

Order	Number	Cumul
0	0	0
1	0	0
2	2	2
3	17	19
4	12	31

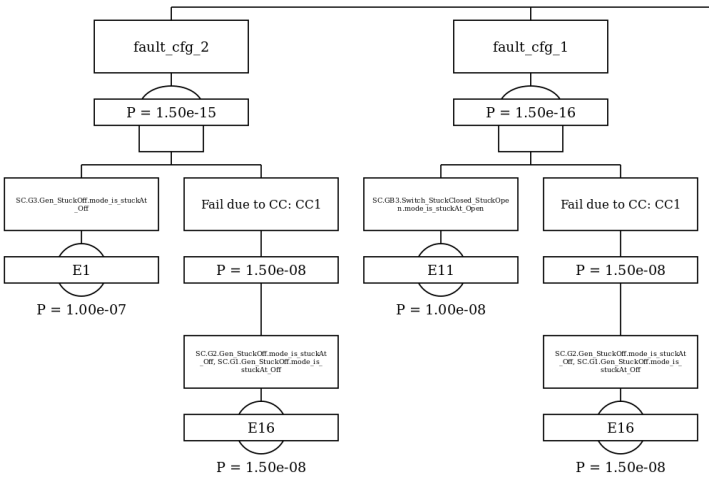

Fig. 16. Statistics and a fault tree for the TMG (excerpt) including a common cause.

of the basic events, without re-generating the fault tree, which may be expensive. It can also be used to perform trade-off analyses, e.g. evaluate the probability of the top level event, while varying the probabilities of (some of) the basic faults. For instance, it may be possible to determine an optimal allocation (w.r.t. some notion of cost) of the reliability of basic components, that guarantees an upper bound of the probability for the top level event.

6.3.2. Common Cause Analysis

We exemplify Common Cause Analysis on the TMG. As an example, Listing 16 shows an example of specification, where the fault extension instructions for the TMG (compare Listing 5) are enriched with the definition of a common cause called CC1. Common causes may have impact on different (instances of) components, e.g. in this example the occurrence of CC1 will cause a failure ('stuck-at-off') of two instances of the generator component, precisely generators G1 and G2 (lines 6 and 7). Failure of the components may be simultaneous or, more in general, be constrained by a given dynamics (*cascading faults*), in this example (line 7) failures will occur within 1 and 2 steps w.r.t. the initiating event (the common cause). Probability is attached (line 4) to the common cause; in this example the probability of the common cause is significantly higher ($1.5 * 10^{-8}$) than the product of the individual probabilities of failure of the two generators ($1 * 10^{-14}$).

We now re-run FTA for the TMG, for the top level event corresponding to the negation of invariant 'R2_false'. Note that in the analysis, common causes are evaluated along with individual events (e.g., generators G1 and G2 may fail as a result of CC1, but also as independent events). All possible combinations of failures are considered in the analysis. The results of the analysis are shown in Fig. 16. In particular, we show the statistics on the minimal cut sets on the left, and an excerpt of the fault tree on the right. As a consequence of the inclusion of the common cause in the analysis, we have additional cut sets. Moreover, the system is no more robust w.r.t. double faults, in fact we have two minimal cut sets of order 2, which are displayed in the excerpt of the fault tree on the right of Fig. 16. The minimal cut sets of order 2 correspond to rows 9 and 14 in Fig. 14 (right); they are obtained by replacing two individual independent faults (G1 stuck off, G2 stuck off) with the common cause CC1. Note that rows 9 and 14 in Fig. 14 are still considered in the analysis (they still appear among the minimal cut sets of order 3). As a result of the inclusion of the common cause, the probability of the top level event increases from $1.36 * 10^{-21}$ to $1.65 * 10^{-15}$.

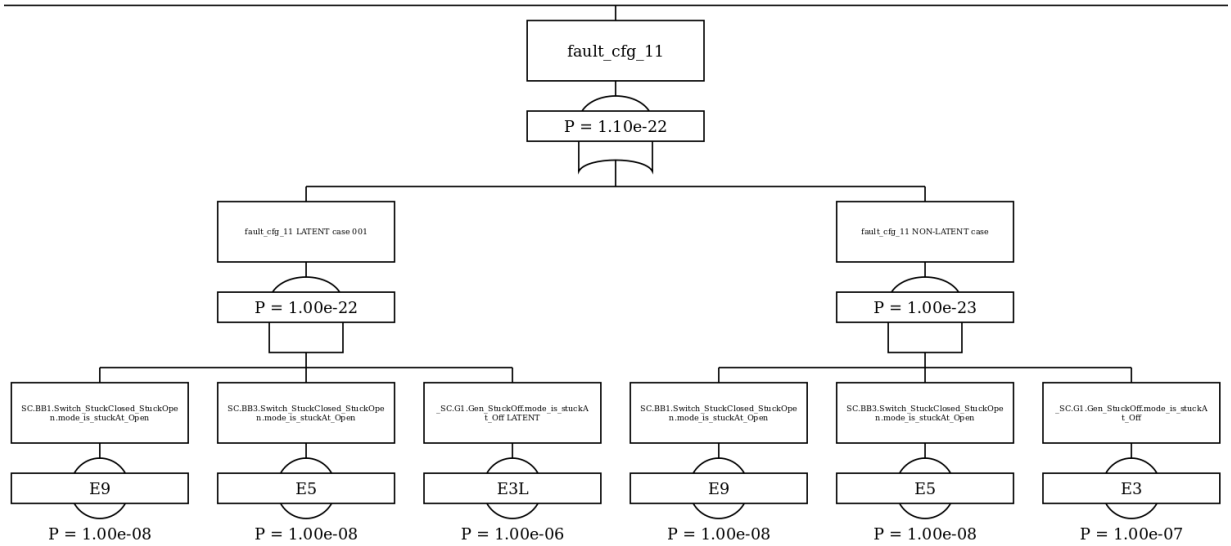

Fig. 17. A fault tree for the TMG (excerpt) including a latent fault.

6.3.3. Latent Faults

We exemplify the specification of latent faults in the TMG example. Let us consider an updated specification for the generators, namely that a generator may fail 'stuck-at-off' latent with a probability of $1 * 10^{-6}$, in addition to the standard probability of $1 * 10^{-7}$. We re-run FTA for the TMG. The probability for the top level event increases from $1.65 * 10^{-15}$ to $1.67 * 10^{-14}$. Fig. 17 contains an excerpt of the generated fault tree, showing the minimal cut set (of order 3) corresponding to row 6 in Fig. 14 (right). The analysis is carried out by splitting cases for latent faults that appear within minimal cut sets. For instance, in the minimal cut set displayed in Fig. 17, one fault (out of three) is possibly latent (the one for generator G1), giving rise to two different cases: one where G1 is failed latent, and one where G1 fails during the mission. Notice that all possible combinations of latent faults are considered in the analysis, e.g. for the minimal cut set corresponding to row 9 in Fig. 14 (right), two components (G1 and G2) may be failed latent, giving rise to a total of 4 cases.

A special treatment is required for minimal cut sets where all faults may be failed latent, e.g., consider row 14 in Fig. 14 (right). In theory, it should give rise to 8 possible combinations. However, following [SAE96], we assume that the combinations where all generators are failed latent cannot occur, therefore only 7 combinations are generated[13]. The rationale of this choice is the assumption that a top level event of a fault tree should be detectable. As a consequence, if all components in a minimal cut sets are failed latent, then the top level event should have been detected before the beginning of the mission, therefore the mission should not have been started. In the case of the TMG we assume that, if all generators are failed 'stuck-at-off', it is detectable that some bus is not powered, hence the system is not put into operation.

7. Fault Detection and Identification Analysis of the TMG

In this section we discuss diagnosability, fault detection and identification analysis. We recall the main definitions and present a general framework encompassing these notions, and we exemplify its application to the TMG model.

[13] This corner case motivates why case splitting is necessary to address FTA in case of latent faults. In fact, just adding up probabilities (latent + standard) and considering only one fault event, would not work.

7.1. Diagnosability, Fault Detection and Identification in xSAP

The design of safety critical systems requires mechanisms to effectively detect and identify faults that may occur during system operation, in order to guarantee safety, or even continue operation in presence of faults. A *diagnoser* is a sub-system whose purpose is to detect and/or identify faults. A diagnoser typically interacts with the system via some sensors (inputs to the diagnoser, also called *observables*) and generates some outputs called *alarms*. An alarm is associated with a *diagnosis condition*, e.g. the detection of a given fault.

The objective of Fault Detection and Identification (FDI) analysis is to assess the quality of a given diagnoser. The quality of the diagnoser may be assessed in terms of correctness and completeness (i.e., an alarm is raised if and only if the corresponding fault has occurred). The existence of a (correct and complete) diagnoser for a given diagnosis condition may depend on the level of observability of the system, i.e. the availability of a set of sensors that can be used for the diagnosis. Diagnosability analysis has the purpose to assess the adequateness of the level of observability, i.e. the possibility to implement a diagnoser for given diagnosis conditions, using the given observables. Namely, a system is diagnosable w.r.t. a diagnosis condition if and only if there exists a (correct and complete) diagnoser for it.

In this paper we follow the approach described in [BCGT14, BCGT15], which describes a formal framework for the design of FDI components.

We assume that a system is given, expressed as an STS $S = \langle V, V_o, W, W_o, F, I, T\rangle$. A diagnosis condition, denoted β, is a Boolean combination of atomic conditions, e.g. a fault (fault identification), or a disjunction of faults (fault detection). A diagnosis condition can be evaluated on any point of a trace representing the execution of the system. Based on the definition of a diagnosis condition, we define an *alarm condition* as the association between a diagnosis condition and the raising of the corresponding alarm. For instance, an alarm condition may prescribe that an alarm condition β is raised within a given *delay* (e.g., 5 time steps) after β has become true. Following [BCGT14], we define three different patterns for alarm conditions, called *exact delay*, *bounded delay* and *finite delay*.

Definition 7.1 (Alarm Condition). Given an alarm A, a diagnosis condition β and a delay $d \geq 0$, we define alarm conditions as the following patterns:

1. EXACTDEL(A, β, d) specifies that whenever β is true, A must be triggered exactly d steps later and A can be triggered only if d steps earlier β was true; formally, for any trace π of the system, if β is true along π at the time point i, then A is true in $\pi[i + d]$ (completeness); if A is true in $\pi[i]$, then β must be true in $\pi[i - d]$ (correctness).
2. BOUNDDEL(A, β, d) specifies that whenever β is true, A must be triggered within the next d steps and A can be triggered only if β was true within the previous d steps; formally, for any trace π of the system, if β is true along π at the time point i then A is true in $\pi[j]$, for some $i \leq j \leq i + d$ (completeness); if A is true in $\pi[i]$, then β must be true in $\pi[j']$ for some $i - d \leq j' \leq i$ (correctness).
3. FINITEDEL(A, β) specifies that whenever β is true, A must be triggered in a later step and A can be triggered only if β was true in some previous step; formally, for any trace π of the system, if β is true along π at the time point i then A is true in $\pi[j]$ for some $j \geq i$ (completeness); if A is true in $\pi[i]$, then β must be true along π in some time point between 0 and i (correctness).

We now define the notion of diagnosability for alarm conditions. Diagnosability is defined relative to a *context*, i.e. a subset of traces of the system. The context is useful to restrict the set of possible executions of the system. For instance, in cases where the occurrence of multiple faults is unlikely, we could restrict to executions with at most one fault. As another example, we could restrict to executions where a given fairness constraint holds, e.g. we can assume that a given component is switched on periodically.

Definition 7.2 (Diagnosability). Let S be a system, β a diagnosis condition and $\Psi \subseteq \Pi_S$ a context. We say that:

1. EXACTDEL(A, β, d) is diagnosable in S iff for any pair of traces σ_1,$\sigma_2 \in \Psi$ and for all $i \geq 0$, if $obs(\sigma_1^{i+d}) = obs(\sigma_2^{i+d})$ then $\sigma_1, i \models \beta$ iff $\sigma_2, i \models \beta$.
2. BOUNDDEL(A, β, d) is diagnosable in S iff for any pair of traces σ_1,$\sigma_2 \in \Psi$ and for all $i \geq 0$ there exists a j, $i \leq j \leq i + d$, s.t. if $obs(\sigma_1^j) = obs(\sigma_2^j)$ and $\sigma_1, i \models \beta$ then $\sigma_2, k \models \beta$ for some k, $j - d \leq k \leq j$.
3. FINITEDEL(A, β) is diagnosable in S iff for any pair of traces σ_1,$\sigma_2 \in \Psi$ and for all $i \geq 0$ there exists a j, $j \geq i$, s.t. if $obs(\sigma_1^j) = obs(\sigma_2^j)$ and $\sigma_1, i \models \beta$ then $\sigma_2, k \models \beta$ for some k, $k \leq j$.

A diagnoser can be formally defined as a (deterministic) STS D which is synchronously composed with the system S. It can be proved that an alarm condition is diagnosable if and only if there exists a diagnoser for it [BCGT14].

The language for alarm conditions in Definition 7.1 is called Alarm Specification Language (ASL); it includes

```
1 NAME: alarm_G1
2 CONDITION: SC.G1.Gen_StuckOff.mode != NOMINAL
3 TYPE: finite
```

Listing 17. An ASL specification for the TMG.

```
1 SC.G1.state
2 CN.cmd_G1
```

Listing 18. An observable specification for the TMG.

the patterns for exact delay, bounded delay and finite delay. The properties of correctness and completeness for ASL patterns can be formalized in LTL (with past operators). For instance, for the BOUNDDEL(A, β, d) pattern, correctness is given by the LTL property $G(A \rightarrow O^{\leq d}\beta)$ whereas completeness is given by $G(\beta \rightarrow F^{\leq d}A)$. Diagnosability, instead, can be expressed in temporal epistemic logic [BCGT14].

XSAP provides a few capabilities related to diagnosability and FDI. First, it enables the diagnosability check for an alarm condition (checking the adequateness of given observables). Second, it is possible to synthesize a set of observables that guarantee diagnosability for an alarm condition (synthesis of observables). Third, it supports the synthesis of a diagnoser for an alarm condition (diagnoser synthesis). Finally, it is possible to assess the quality of a diagnoser (effectiveness analysis).

7.2. Diagnosability, Fault Detection and Identification Analysis of the TMG

We now exemplify the diagnosability and FDI functionalities of XSAP on the TMG case study. In the model described in Section 5.1, the controller can directly observe component faults (and repairs), therefore it has complete observability over component faults. Here we are interested in analyzing the observability requirements of the TMG before the controller is plugged into the system. Therefore, we start from a model of the system with an empty controller (we keep only the interface of the module[14]).

We first illustrate diagnosability analysis. We consider, for instance, diagnosing a generic fault for a generator, e.g. G1. Diagnosability takes as input an ASL specification and the list of observables. Example specifications are shown in Listing 17 and Listing 18. The ASL specification defines an alarm called *alarm_G1* that is associated to an alarm condition, of type *finite delay*, specifying that G1 is not in nominal mode. The list of observables includes the state of the generator and the command sent to G1 by the controller. In other words, we are checking whether is it possible to diagnose a fault of generator G1 eventually, with the given observables.

Running XSAP, the alarm condition is found to be non-diagnosable, and a pair of traces is returned as counterexample. Intuitively, the pair is such that the two traces are observationally indistinguishable, however the fault manifests itself in one trace and not in the other. Formally, according to Definition 7.2, we have a pair of traces σ_1,σ_2 and an index i such that $obs(\sigma_1) = obs(\sigma_2)$, $\sigma_1, i \models \beta$ and $\sigma_2, j \not\models \beta$ for all j. Such pair is called *critical pair* [BCGT15]. Fig. 18 shows the critical pair generated by XSAP, displaying a subset of the signals. The analysis of the critical pair reveals that the alarm condition is not diagnosable, since generator G1 is never commanded to switch on. Since the generator is never switched on, its fault has no effect on the state of the generator itself.

We can prove that the alarm condition is diagnosable under a more restrictive context. As an example, we add the following line: '**CONTEXT**: G F [0,5] CN.cmd_G1 = cmd_on' to the ASL specification in Listing 17. That is, we assume that, in any execution, the generator is commanded to switch on within at most 5 steps. With this fix, the alarm condition is found to be diagnosable. In fact, we can prove a stronger property, namely that the alarm condition is bounded delay diagnosable, with a delay of 5 steps, under the given context. Delay 4, instead, generates a critical pair, as expected.

We now exemplify the synthesis of observables. XSAP supports the generation of minimal sets of observables, that guarantee diagnosability. The inputs to the analysis are the same as for diagnosability. This time, however, we list as (potential) observables all the variables representing the internal state of components, and the powering information for buses, for a total of 45 signals. Running XSAP, we obtain a set of observables of cardinality 1, shown in Fig. 19. Namely, the alarm condition is diagnosable, and a diagnoser can be built by observing the internal state of G1. Note that the powering information for buses cannot be used to infer the status of the generator, since when the generator is commanded to switch on, we have no guarantee that the bus is connected to the generator. However, if we restore the constraints in Listing 14 and the definition of the monitor in the controller module, then 'SC.B1_poweredby_G1_U' and 'SC.B1_poweredby_G1' can be shown to be alternative sets of observables, since in this case we can infer the status of G1 by observing the consequences on bus B1. Furthermore, note that under the given context, it is not necessary to

[14] The models used in this section are available in the XSAP distribution under examples/FDI.

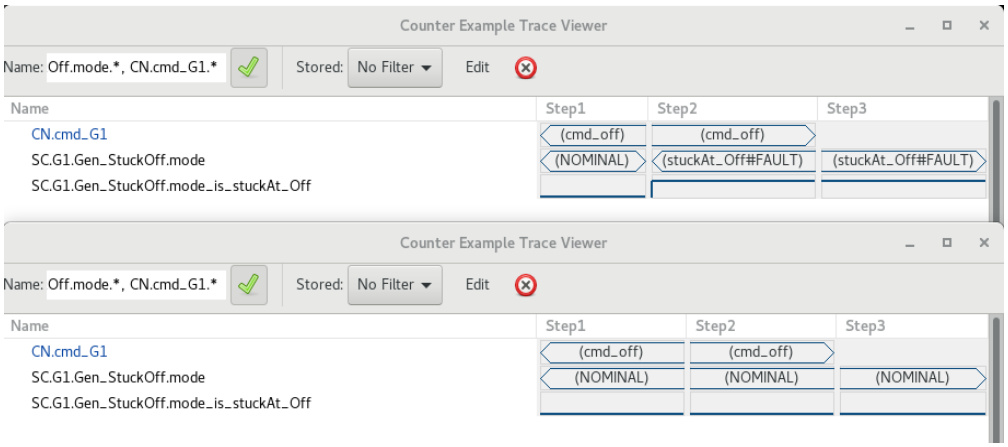

Fig. 18. Critical pair for the TMG.

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

Fig. 19. Outcome of observables synthesis for the TMG.

observe the command sent to G1, since we rely on the fact that the command cmd_on is guaranteed to be sent to G1 within at most 5 steps.

The result of the observables synthesis can be used as guidance to design a diagnoser. For instance, it is possible to choose the set of observables needed for the diagnosis, depending on cardinality and/or cost considerations. In the case of generator G1 for the TMG, we can choose to monitor either its internal state (*on* or *off*) or the power connection that feeds B1 (in the scenario where 'SC.B1_poweredby_G1_U' and 'SC.B1_poweredby_G1' are possible observables). This corresponds to putting a sensor on either G1 or the connection from G1 to B1. Monitoring such sensor makes the model more realistic w.r.t. the model where the diagnoser directly observes internal faults in the components.

In the rest of this section, instead of modeling the diagnoser by hand, we use XSAP to illustrate how the diagnoser can be synthesized automatically. Diagnoser synthesis in XSAP takes as input the model, the list of observables and an ASL specification, and generates a diagnoser model, which is automatically integrated, i.e. synchronously composed, with the original model. In particular, the diagnoser is implemented as an SMV module, which is instantiated in the main module. We run XSAP and synthesize an FDI model for the generic fault of generator G1, using 'SC.G1.state' as observable, and for an ASL specification with context and bounded delay with delay 5 (as for diagnosability analysis). Running XSAP, we automatically obtain the integrated model including the alarm 'alarm_G1'. Listing 19 shows the generated FDI module. The synthesis algorithm [BCGT14] generates an automaton that encodes the set of possible states (called *belief states*) in which the system may be after each observation. The diagnoser is obtained by annotating each belief state with the information on the alarm being satisfied or not. In particular, the signal 'Kalarm_G1' (respectively 'Knalarm_G1' and 'Ualarm_G1') annotates states where the diagnoser knows that the alarm must be raised (respectively: knows that the alarm must not be raised; it is unknown whether the alarm must be raised).

The generated diagnoser can be checked against a set of properties to assess its quality (effectiveness analysis). For instance, we can check completeness of the diagnoser with this property: '(G F [0,5] CN.cmd_G1 = cmd_on) -> (G (SC.G1.Gen_StuckOff.mode != NOMINAL -> (F [0,5] __myfdir.myfd.Kalarm_G1)))', which is proved true by XSAP.

8. Experimental Evaluation

In this section we experimentally evaluate the performance of XSAP on the TMG model, focusing on the FTA functionality. In order to evaluate the scalability of the tool, we have generalized the TMG model and made it parametric in

```
MODULE __FD("SC.G1.state")
 VAR
  __state : 1 .. 27;
 DEFINE
  __expr14 := next(__state) = 15;
  __expr15 := next(__state) = 4;
  __expr13 := next(__state) = 16;
  Kalarm_G1 := ((((((((((__state = 11 | __state = 26) | __state = 24) | __state = 23) | __state = 13) | __state = 9) | __state = 21) | __state = 25) |
    __state = 22) | __state = 14) | __state = 12) | __state = 10);
  Ualarm_G1 := (((((((__state = 8 | __state = 18) | __state = 6) | __state = 3) | __state = 20) | __state = 16) | __state = 5) | __state = 7) |
    __state = 19) | __state = 17);
  Knalarm_G1 := (((__state = 2 | __state = 15) | __state = 1) | __state = 4);
  __sink_state := __state = 27;
 INIT (__state = 1 | __state = 27);
 INVAR (!(__state = 27) | !("SC.G1.state" = on));
 INVAR (!(__state = 1) | "SC.G1.state" = on);
 TRANS case
  next("SC.G1.state") = off : case
   (__state = 26 | __state = 25) : next(__state) = 26;
   __state = 24 : next(__state) = 25;
   __state = 23 : next(__state) = 24;
   __state = 22 : next(__state) = 23;
   __state = 21 : next(__state) = 22;
   __state = 20 : next(__state) = 21;
   __state = 19 : next(__state) = 20;
   __state = 18 : next(__state) = 19;
   __state = 17 : next(__state) = 18;
   __state = 16 : next(__state) = 17;
   __state = 15 : __expr13;
   (__state = 14 | __state = 13) : next(__state) = 14;
   __state = 12 : next(__state) = 13;
   __state = 11 : next(__state) = 12;
   __state = 10 : next(__state) = 11;
   __state = 9 : next(__state) = 10;
   __state = 8 : next(__state) = 9;
   __state = 7 : next(__state) = 8;
   __state = 6 : next(__state) = 7;
   __state = 5 : next(__state) = 6;
   __state = 4 : __expr13;
   __state = 3 : next(__state) = 5;
   __state = 2 : __expr13;
   TRUE : (__state = 1 & next(__state) = 3);
  esac;
  TRUE : (next("SC.G1.state") = on & case
   (__state = 20 | (__state = 19 | (__state = 18 | (__state = 17 | (__state = 16 | __state = 15))))) : __expr14;
   (__state = 8 | (__state = 7 | (__state = 6 | __state = 5))) : __expr15;
   __state = 4 : __expr14;
   __state = 3 : __expr15;
   __state = 2 : __expr14;
   TRUE : (__state = 1 & next(__state) = 2);
  esac);
esac;
```

Listing 19. A synthesized FDI model for the TMG.

the number of components. Fig. 20 shows a parametric version of the TMG model with n buses, i.e with n generators and $2n$ circuit breakers[15], where $n \geq 3$.

The requirements of the TMG can be generalized to the parametric model. In particular, requirement R3 (Bus power source priority and source to bus path priority schemes) must be generalized to encompass the additional paths of the system. There are several possible ways to generalize requirement R3. Out of the different possibilities, we chose one, which we illustrate in Fig. 21 and 22 for the case where $n = 5$ (note that this extension is conservative, i.e.

[15] In practical applications, it might be worth considering a more complex generalization of the model, namely one could increase the redundancy of the bus connections, in order to increase the reliability of the overall system. However, the current model is sufficient for our purposes, namely scalability evaluation.

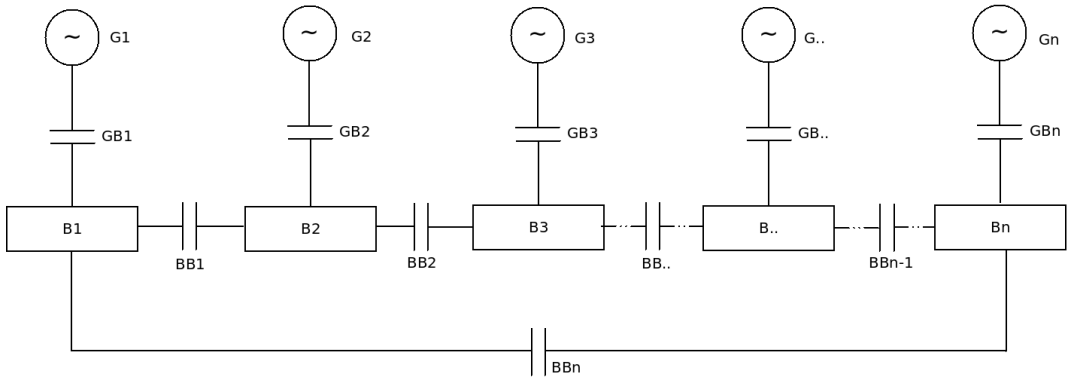

Fig. 20. Triple Modular Generator: parametric model.

BUS	Highest priority	...	...	...	Lowest priority
B1	G1	G2	G3	G4	G5
B2	G2	G1	G3	G4	G5
B3	G2	G1	G4	G5	G3
B4	G2	G1	G3	G5	G4
B5	G2	G1	G3	G4	G5

Fig. 21. Bus power source priority ($n = 5$).

the case $n = 3$ is the same as described in Section 3). Specifically, we enforce the following properties. In absence of faults, only two generators are switched on, namely G1 and G2, B1 is powered by G1 and all other buses by G2. For buses B3, B4 and B5, powering via the corresponding generator (respectively, G3, G4 and G5) is the lowest priority option. The remaining options, for all buses, are ordered from left to right (i.e., Gj has higher priority than Gk if and only if $j < k$). Given a bus and a generator, there are two possible paths connecting them, and we choose the high priority path to be the shortest one[16]. The other requirements (R1, R2, R4, R5, R6) are unchanged[17].

We have implemented a generator, written in Python, that automatically generates the (nominal) SMV models, given the parameter n. The corresponding extended models can be generated automatically as described in Section 5.3. Note that the FEI specifications are unchanged, since they are defined on module types rather than on module instances[18]. In the rest of this section we consider the models extended with the fault extension instructions in Listing 5.

The generator as well as the generated (nominal) models for $n = 3, \ldots, 20$ can be found in the XSAP distribution under examples/fe/triple_modular_generator/parametric_models.

[16] In the case where n is even and the two paths have the same length, we conventionally chose either the right or the left connection to be the high priority path.

[17] Similarly to the case where $n = 3$, it can be shown that requirements R4, R5 and R6 also hold for the generalized models.

[18] The FEI specification may be affected if common causes are considered, since common causes can be defined on component instances. Here we do not consider common causes.

Paths	Priority	B1	B2	B3	B4	B5
G1	High	—	BB1	BB2-BB1	BB4-BB5	BB5
	Low	—	BB2-BB3-BB4-BB5	BB3-BB4-BB5	BB3-BB2-BB1	BB4-BB3-BB2-BB1
G2	High	BB1	—	BB2	BB3-BB2	BB5-BB1
	Low	BB5-BB4-BB3-BB2	—	BB3-BB4-BB5-BB1	BB4-BB5-BB1	BB4-BB3-BB2
G3	High	BB1-BB2	BB2	—	BB3	BB4-BB3
	Low	BB5-BB4-BB3	BB1-BB5-BB4-BB3	—	BB4-BB5-BB1-BB2	BB5-BB1-BB2
G4	High	BB5-BB4	BB2-BB3	BB3	—	BB4
	Low	BB1-BB2-BB3	BB1-BB5-BB4	BB2-BB1-BB5-BB4	—	BB5-BB1-BB2-BB3
G5	High	BB5	BB1-BB5	BB3-BB4	BB4	—
	Low	BB1-BB2-BB3-BB4	BB2-BB3-BB4	BB2-BB1-BB5	BB3-BB2-BB1-BB5	—

Fig. 22. Source to bus path priority ($n = 5$).

n	State vars	Input vars	Total	Nr. bits
3	48	42	90	123
4	64	56	120	164
5	80	70	150	205
6	96	84	180	246
7	112	98	210	287
8	128	112	240	328
9	144	126	270	369
10	160	140	300	410
11	176	154	330	451
12	192	168	360	492
13	208	182	390	533
14	224	196	420	574
15	240	210	450	615
16	256	224	480	656
17	238	272	510	697
18	288	252	540	738
19	304	266	570	779
20	320	280	600	820

Table 1. TMG parametric models: statistics.

n	Card	# MCS	PARAM		BMC-PARAM	
			T(sec)	M(Gb)	T(sec)	M(Gb)
3	4	26	3.03	0.06	0.94	0.06
4	5	72	17.35	0.09	2.85	0.07
5	6	182	92.41	0.12	12.63	0.09
6	7	436	3360.28	0.28	327.82	0.11
7	8	1010	TO	-	24012.97	0.18
8	-	-	TO	-	TO	-
9	-	-	TO	-	TO	-
10	-	-	TO	-	TO	-

Table 2. TMG parametric models: FTA for TLE 'R2_false'.

Table 1 lists some statistics for TMG models (the extended models, i.e. those that are run in the experiments), for $n = 3, \ldots, 20$. The table lists the number of state variables, the number of input variables (including fault variables and controller commands), the total number of variables (state + input variables) and the total number of bits (i.e., the total number of Boolean variables in the encoded model). The number of bits is higher than the total number of variables, since some variables are enumerative and encoded with 2 or more bits.

The experiments have been run on a cluster of Linux machines running Scientific Linux 7.5, and equipped with 2.67 GHz Intel Xeon CPUs. In the rest of this section, we use XSAP to generate fault trees. We use two different algorithms. The first algorithm (referred to as PARAM from now on) is based on parameter synthesis and IC3, and is described in [BCMG15]. It computes minimal cut sets by increasing order (cardinality layers). The second algorithm (referred to as BMC-PARAM from now on) uses BMC to collect the minimal cut sets up to a given depth (i.e., the depth is the length of the encoded trace for BMC search) and then uses IC3 to close the induction (as in the first algorithm). The second algorithm exploits the fact that the TMG model has a shallow depth, in particular minimal cut sets can be found by searching up to a depth $k = 2$. In both cases, we set the option 'boolean_conversion_uses_predicate_normalization', which reduces the memory consumption.

In the first experiment, we evaluated the models for n up to 10 and ran FTA for the top level event corresponding to the negation of invariant 'R2_false' (i.e., stating that at least one bus is not powered), as done in Section 6.3. We set a time out of 3 days and a memory limit of 50Gb. The results are collected in Table 2. For each model, we list the maximum cardinality of minimal cut sets, the total number of minimal cut sets and, for both the PARAM and BMC-PARAM algorithms, the running time (in seconds; 'TO' stands for time out) and the maximum amount of used memory (in Gb; 'MO' stands for memory out). The experiments show that running time rapidly grows with n. The maximum cardinality of the minimal cut sets also grows with n (empirically, we observe that the maximum cardinality is $n + 1$ – this is fully proved for n up to 7). Clearly, the increase in the running time is justified by the dimension of the models and the potential combinations of cut sets that have to be considered (for dimension n, the minimal cut sets include combinations of at most $n + 1$ faults out of a total number of $5n$ faults; for instance, for $n = 7$, this is more

n	Card	# MCS	PARAM		BMC-PARAM	
			T(sec)	M(Gb)	T(sec)	M(Gb)
3	4	18	2.68	0.06	0.92	0.06
4	5	50	9.79	0.08	2.22	0.07
5	6	130	49.66	0.11	4.96	0.08
6	7	322	477.03	0.19	23.93	0.10
7	8	770	8503.48	0.42	124.29	0.14
8	9	1794	224468.26	1.17	99.22	0.19
9	10	4098	TO	-	1415.11	0.35
10	11	9218	TO	-	760.80	0.94
11	12	20482	TO	-	5170.52	2.74
12	13	45058	TO	-	7575.87	8.87
13	14	98306	TO	-	9806.41	29.98
14	-	-	TO	-	-	MO
15	-	-	TO	-	-	MO

Table 3. TMG parametric models: FTA for TLE 'B1 is not powered'.

than $3 * 10^7$ possible combinations). From the results, we can see that BMC-PARAM performs consistently better than PARAM and is able to complete the case $n = 7$, for which PARAM times out.

In the second experiment, we want to analyze the impact of the TLE on the FTA computation. In particular, in the first experiment the TLE ('At least one bus is not powered') is a disjunctive formula, hence some of the generated cut sets are logically analogous, due to symmetries, i.e. they represent similar faulty configurations that affect a different bus. In the second experiment we use the TLE '!SC.B1.is_powered' ('Bus B1 is not powered'). Table 3 collects the results. This time we run experiments for models up to $n = 15$. The results shows that for this TLE, the running time decreases w.r.t. the first experiment, along with the number of generated minimal cut sets (for a given n). More in detail, algorithm PARAM is able to complete the analysis up to $n = 8$, whereas BMC-PARAM is able to reach $n = 13$, computing more than 98.000 minimal cut sets. As in the first experiment, BMC-PARAM performs consistently better than PARAM. The improvement in performance in the second experiment, opens the opportunity to optimize the generation of minimal cut sets for a disjunctive top level such as the one in the first experiment. In particular, a possible strategy would be to separately generate minimal cut sets for the n top level events 'Bus B_i is not powered', for $i = 1, \ldots, n$, and then combine the results (the combination would imply removing duplicated and possibly non-minimal cut sets). The results in Table 3 suggest that this strategy may be effective. We leave this investigation as future work.

In the third experiment, we run FTA again for the TLE corresponding to the negation of invariant 'R2_false ('At least one bus is not powered'), but this time we limit the maximum cardinality of the minimal cut sets that we want to compute. Limiting the maximum cardinality of the cut sets is usually justified in standard practice, since high-order cut sets typically have very low probabilities and can be disregarded, since the contribution to the overall probability of the TLE is marginal. Indeed, in standard practice it is infrequent to compute minimal cut sets for orders higher than 4. Using XSAP, we can formally guarantee the above argument ('The contribution to the overall probability of the TLE of high-order cut sets is marginal'). In fact, using the *anytime computation* functionality described in [BCMG15], we can compute a lower and an upper bound for the probability of the TLE after completing each cardinality layer. The upper bound provides us an over-approximation of the probability of the TLE, and by comparing the lower and the upper bound (under- and over- approximations) we can evaluate the potential error (the quality of the approximation).

We run XSAP for models up to $n = 20$, using anytime computation for a maximum cardinality of 4. We set a time out of 3 days and a memory limit of 100Gb. The results are collected in Table 4. The table lists the number of cut sets found for cardinalities 3 and 4, the total number of cut sets, the upper bound probability, the worst-case error (computed as the difference between the upper and the lower bound) and, for both the PARAM and BMC-PARAM algorithms, the running time (in seconds) and the memory consumption (in Gb) As we can see from the results, the worst-case error is negligible w.r.t. the upper bound probability in all cases (excluding MO cases), in fact the ratio between the error and the probability ranges between 10^{-14} and 10^{-8} for cases $n = 3, \ldots, 18$. Note that for $n = 19$ and $n = 20$ (MO cases), the error is computed based on the cut sets of order 3, since layer 4 is not guaranteed to be complete – even in those cases, the error is acceptable, since it is lower than 1%. From the results, it is also clear that BMC-PARAM performs better in time than PARAM, except in a few cases with lower cardinalities, although at a price of a higher memory consumption.

If we compare the outcome of the third experiment w.r.t. the first experiment (sharing the same TLE), as expected, we can see that the computation time significantly improves in the third experiment, where we limit the maximum cardinality of the minimal cut sets. Indeed, we are able to compute a precise over-approximation [BCMG15]of the

n	CS3	CS4	Total	Prob (upper bound)	Error	PARAM		BMC-PARAM	
						T(sec)	M(Gb)	T(sec)	M(Gb)
3	14	12	26	$1.3639999670e^{-21}$	$5.45e^{-36}$	2.90	0.06	0.90	0.06
4	8	32	40	$4.4000150850e^{-23}$	$7.73e^{-35}$	9.50	0.08	12.75	0.09
5	10	20	30	$5.5000005550e^{-23}$	$4.29e^{-34}$	30.52	0.11	27.64	0.11
6	12	24	36	$6.6000006661e^{-23}$	$1.32e^{-33}$	67.11	0.15	64.38	0.15
7	14	28	42	$7.7000007773e^{-23}$	$3.32e^{-33}$	183.28	0.23	147.13	0.23
8	16	32	48	$8.8000008887e^{-23}$	$7.22e^{-33}$	372.61	0.28	376.68	0.32
9	18	36	54	$9.9000010004e^{-23}$	$1.42e^{-32}$	691.38	0.38	787.55	0.42
10	20	40	60	$1.1000001113e^{-22}$	$2.56e^{-32}$	1778.43	0.52	1550.48	0.58
11	22	44	66	$1.2100001225e^{-22}$	$4.35e^{-32}$	3142.81	0.87	2306.01	0.88
12	24	48	72	$1.3200001339e^{-22}$	$7.02e^{-32}$	6325.30	1.08	5475.07	1.38
13	26	52	78	$1.4300001454e^{-22}$	$1.09e^{-31}$	16029.83	2.00	7218.19	2.18
14	28	56	84	$1.5400001570e^{-22}$	$1.63e^{-31}$	28712.51	2.86	13716.65	3.24
15	30	60	90	$1.6500001689e^{-22}$	$2.36e^{-31}$	40890.66	4.08	32527.51	6.61
16	32	64	96	$1.7600001809e^{-22}$	$3.34e^{-31}$	76266.90	8.53	41899.93	11.45
17	34	68	102	$1.8700001933e^{-22}$	$4.61e^{-31}$	124112.16	16.99	75041.16	23.32
18	36	72	108	$1.9800002061e^{-22}$	$6.24e^{-31}$	237378.51	39.46	123761.47	46.68
19	38	-	-	$2.1075725385e^{-22}$	$1.76e^{-24}$	-	MO	-	MO
20	40	-	-	$2.2217644002e^{-22}$	$1.18e^{-24}$	-	MO	-	MO

Table 4. TMG parametric models: *anytime computation* of MCS for TLE 'R2_false' and maximum cardinality 4.

probability of the TLE and all the minimal cut sets up to cardinality 4, for all models up to $n = 18$, whereas in the first experiment we time out for $n \geq 8$. These results, along with the estimation of the worst-case error, show the effectiveness of the anytime computation of XSAP and the feasibility of automated FTA for models of significant (realistic) size (e.g., the model for $n = 18$ is encoded with more than 700 Boolean variables, and it includes 90 different faults).

9. Related Work

In this section we discuss some related work.

9.1. Model-Based Safety Assessment: Process, Tools and Applications

This work is based on the XSAP platform for safety analysis. XSAP is based on the concepts of automated fault injection and model extension, and on the idea of Model-Based Safety Assessment (MBSA). These ideas have been originally presented in works such as [BVÅ03, JMWH05, BV10, BCK+11].

The FSAP [BV07] platform is the predecessor of XSAP. Originally, FSAP has been developed in the following European projects: ESACS [BVÅ03], ISAAC [ABB+06], and MISSA [MIS15]. XSAP is a full re-implementation of FSAP, and it provides numerous extensions. Among them, the support for infinite-state systems, extended and customizable libraries to define fault modes and fault dynamics, diagnosability and fault detection and identification analysis. In addition, XSAP provides novel routines for fault tree generation, namely the original BDD-based routines [BCT07] are complemented by SAT-based and SMT-based routines, including routines based on IC3 [BCMG15].

Recent industrial applications of MBSA are described in, e.g. [BCC+03, ANY+12, GFB+14, BCK+14, BCP+15]. XSAP is the core verification engine for many other tools. It has been used in several industrial projects funded by the European Space Agency [Eur07, Eur10, Eur11, Eur13a, Eur13b]. Finally, XSAP is currently being used in a joint research and development project, in the avionics domain, between FBK and The Boeing Company [BCP+15] and in another joint project between FBK and Bosch, in the automotive domain.

XSAP is also used as a back-end for the COMPASS toolset [BCK+11, BCK+14], see also [BBC17]. In COMPASS the fault models must be modeled manually using the SLIM language (a variant of AADL), whereas in XSAP the model extension is based on a customizable library for defining fault modes and their dynamics.

Examples of alternative platforms for MBSA are based on Altarica/OCAS [BCS02, PBB+13, BCL+15], Scade [DÅ04, AH05], and Statemate [PCV+06]. Generally, these tools support a subset of the features implemented in XSAP (FTA, FMEA, or some limited form of model extension).

Regarding scalability of the the routines implemented in XSAP, we refer the reader to [BCL+15] for a comparison

with Altarica/OCAS (carried out using a license courtesy of Dassault Aviation), and to [BCMG15] for an exhaustive evaluation of the routines based on IC3.

9.2. Fault Tree Analysis

MBSA tools, including XSAP, have been used to automate the generation of traditional artifacts of safety assessment such fault trees and FMEA tables. The focus on these artifacts is due to the fact that they are used for system certification in many industries, for instance in the avionics and aerospace sectors [SAE96, ECS].

XSAP implements fault tree generation routines based on BDDs [BCT07] and on SAT/SMT [BCMG15]. The latter are based on IC3 [BCMG15] and parameter synthesis [CGMT13], and implement a layered computation of the Minimal Cut Sets (by cardinality). A similar layered approach is presented in [ADS+04], however it is based on bounded model checking and it does not address the problem of convergence.

Hip-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) [Pap00, PM01] is a framework that supports mechanical synthesis of fault trees, based on the structure of the design model. The user has to manually identify the component failure modes and model the local failure behaviour using a tabular technique. The focus of the Hip-HOPS lies in organizing fault trees in accordance with the structure of the design model, rather than in generating fault trees automatically from a behavioral model, as in XSAP.

In [BCMT14], the authors present an approach to build hierarchical fault trees following the system structure. The approach has been implemented in a contract-based framework implemented in the ocra platform [ocr19]. In this context, faults are represented by contract violations. A different approach to generate hierarchical fault trees is based on retrenchment [BB13a, BB13b]. This framework focuses on the relations between nominal and faulty behaviors, and does not address implementation issues.

9.3. Diagnosability, Fault Detection and Identification Analysis

For diagnosability, fault detection and identification we refer to the formal framework described in [BCGT14, BCGT15] ([BCGT14] focuses on synchronous systems, whereas [BCGT15] extends the framework to the asynchronous case). This work proposes a *pattern based* language for the specification of diagnosis and alarm conditions, and supports different forms of delay (exact, bounded and finite). Additional aspects accommodated in the framework are *diagnosability* [SSL+96], the notion of *maximality* of the diagnoser, and the notion of *trace diagnosability*. The latter enables reasoning on systems that are only locally diagnosable and extends the previous work such as [CPC03]. This framework has been evaluated in several projects funded by the European Space Agency, such as AUTOGEF [ANY+12] and FAME [GFB+14, BBC17], on a case study based on the EXOMARS Trace Gas Orbiter.

The pattern-based language of [BCGT14, BCGT15] is expressed in epistemic temporal logic [HV89]. It is called Alarm Specification Language with Epistemic operators (ASL_K). Here, the *knowledge* operator is used to express the certainty of a condition, based on given observations. Diagnosis and alarm conditions are expressed in LTL with past operators, whereas epistemic logic is needed to formalize the concepts of trace diagnosability and maximality. We refer the reader to [ELMV11] and [Hua13] for alternative approaches that formalize diagnosability in epistemic logic. However, these works are limited to finite-delay diagnosability, and do not consider notions such as trace diagnosability.

Within the framework of [BCGT14, BCGT15], it is possible to validate a given set of diagnosability requirements [CRST12] and to verify whether a candidate diagnoser satisfies a given set of requirements. Verification and validation can be performed using a a model checker for temporal epistemic logic such as MCK [GVDM04] or using a model checker for LTL (in case the specification falls into the LTL fragment). The framework also addresses the automated synthesis of the diagnoser based on a construction inspired by [SSL+96] and Schumann [Sch04]. Finally, [BBCO12] addresses the problem of synthesizing cost-optimal sets of observations.

For a review of some recent literature on fault management systems, diagnosability, fault detection and identification, and the relations with the framework presented in [BCGT14, BCGT15], we also refer the reader to [Boz17].

9.4. Analysis of Power Systems

Evolution to a More Electric Aircraft (MEA) [Ray18] has a significant impact on the design of electrical power systems. These systems tend to have more generators and a more highly redundant distribution system. Similar considerations apply to the electrical power industry. Manual analysis, based on the explicit enumeration of the faulty

configurations, for such systems is impractical and error-prone, if not impossible, making the use of formal, automated techniques, highly recommended.

A recent effort in this direction is the formulation of a challenging case study [Bou17], to be used as a benchmark for modeling and verification technologies. The case study is an emergency power supply system for a nuclear power plant, with redundancy characteristics and repairable components, and complex failures (in function and on demand, common cause failures). In [Bou17] the case study is formalized as a reliability model based on BDMP (Boolean logic Driven Markov Processes), and some partial reliability results are provided. Recently, a workshop has been organized [EDF19], to present and compare different tools and methodologies applied to the case study.

10. Conclusions and Future Work

In this paper we described an industrial case-study that represents a redundant on-board power supply with reconfiguration policy. The case-study is non-trivial from both the modeling and the verification viewpoint, due to the presence of complex correctness and priority requirements for the controller, and to the high number of faulty configurations that have to be dealt with.

We explained in detail the modeling aspects of the case study, and provided a general modeling strategy that addresses the implementation of the requirements and the automatic synthesis of a controller that satisfies them by construction. In particular, our model implements a highly complex reconfiguration policy, and is able to address in a principled way the dependencies between the different requirements for the controller (namely, it solves conflicts between admissible powering options for different buses). We think that this modeling strategy is very general, and can be adapted to a variety of case studies that feature similar characteristics.

We applied our modeling and verification to a parametric version of the case study, and showed that our verification tools scale to models of much bigger size, making our methodology suitable for the analysis of modern systems in the aerospace and energy sectors.

As part of our future work, we plan to extend our approach to hybrid systems, and apply it to a challenging case study from the nuclear power sector [Bou17] and to relay-based circuits in the railway domain. Preliminary work has been done in modeling similar systems using Switched Multi-Domain Linear Kirchhoff Networks [CMS17, CCM+18] and analyzing them using SMT. We also intend to extend our technology with techniques such as partitioning and abstraction, in order to improve the scalability even further. Finally, we would like to extend XSAP by incorporating other analyses such as architectural trade studies, carried out in the early phases of development [BCM19].

References

[ABB+06] O. Akerlund, P. Bieber, E. Böde, M. Bozzano, M. Bretschneider, C. Castel, A. Cavallo, M. Cifaldi, J. Gauthier, A. Griffault, O. Lisagor, A. Ludtke, S. Metge, C. Papadopoulos, T. Peikenkamp, L. Sagaspe, C. Seguin, H. Trivedi, and L. Valacca. ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects. In *Proc. ERTS*, January 2006. Tolouse.

[ADS+04] P.A. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, and O. Åkerlund. Designing Safe, Reliable Systems Using Scade. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, First International Symposium, ISoLA 2004, Paphos, Cyprus, October 30 - November 2, 2004, Revised Selected Papers*, volume 4313 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2004.

[AE90] E. Allen Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science, 1990.

[AH05] A.Joshi and M.P.R Heimdahl. Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In R.Winther, B.A. Gran, and G. Dahll, editors, *Computer Safety, Reliability, and Security, 24th International Conference, SAFECOMP 2005, Fredrikstad, Norway, September 28-30, 2005, Proceedings*, volume 3688 of *Lecture Notes in Computer Science*, pages 122–135. Springer, 2005.

[ANY+12] E. Alaña, H. Naranjo, Y. Yushtein, M. Bozzano, A. Cimatti, M. Gario, R. de Ferluc, and G. Garcia. Automated generation of FDIR for the compass integrated toolset (AUTOGEF). In *DASIA*, volume ESA SP 701, 2012.

[BB13a] R. Banach and M. Bozzano. The Mechanical Generation of Fault Trees for Reactive Systems via Retrenchment I: Combinational Circuits. *FAC*, 25(4):573–607, 2013.

[BB13b] R. Banach and M. Bozzano. The Mechanical Generation of Fault Trees for Reactive Systems via Retrenchment II: Clocked and Feedback Circuits. *FAC*, 25(4):609–657, 2013.

[BBC+16] B. Bittner, M. Bozzano, R. Cavada, A. Cimatti, M. Gario, A. Griggio, C. Mattarei, A. Micheli, and G. Zampedri. The xSAP Safety Analysis Platform. In *Proc. TACAS*, volume 9636 of *LNCS*, pages 533–539, 2016.

[BBC17] Benjamin Bittner, Marco Bozzano, and Alessandro Cimatti. Timed failure propagation analysis for spacecraft engineering: The ESA solar orbiter case study. In Marco Bozzano and Yiannis Papadopoulos, editors, *Model-Based Safety and Assessment - 5th International Symposium, IMBSA 2017, Trento, Italy, September 11-13, 2017, Proceedings*, volume 10437 of *Lecture Notes in Computer Science*, pages 255–271. Springer, 2017.

[BBCO12] B. Bittner, M. Bozzano, A. Cimatti, and X. Olive. Symbolic Synthesis of Observability Requirements for Diagnosability. In *AAAI Conference on Artificial Intelligence*, 2012.

[BCC+03] M. Bozzano, A. Cavallo, M. Cifaldi, L. Valacca, and A. Villafiorita. Improving Safety Assessment of Complex Systems: An Industrial Case Study. In *FME*, volume 2805 of *LNCS*, pages 208–222, 2003.

[BCGT14] M. Bozzano, A. Cimatti, M. Gario, and S. Tonetta. Formal Design of Fault Detection and Identification Components Using Temporal Epistemic Logic. In *Proceedings of TACAS'14*, pages 326–340, Grenoble, France, 2014.

[BCGT15] M. Bozzano, A. Cimatti, M. Gario, and S. Tonetta. Formal Design of Asynchronous Fault Detection and Identification Components using Temporal Epistemic Logic. *Logical Methods in Computer Science*, 11(4): 2015.

[BCK+11] M. Bozzano, A. Cimatti, J.-P. Katoen, V.Y. Nguyen, T. Noll, and M. Roveri. Safety, Dependability and Performance Analysis of Extended AADL Models. *Comput. J.*, 54(5):754–775, 2011.

[BCK+14] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Panagiotis Katsaros, Konstantinos Mokos, Viet Yen Nguyen, Thomas Noll, Bart Postma, and Marco Roveri. Spacecraft early design validation using formal methods. *Rel. Eng. & Sys. Safety*, 132:20–35, 2014.

[BCL+15] M. Bozzano, A. Cimatti, O. Lisagor, C. Mattarei, S. Mover, M. Roveri, and S. Tonetta. Safety Assessment of AltaRica Models via Symbolic Model Checking. *Science of Computer Programming*, 98(4):464–483, 2015.

[BCM19] Marco Bozzano, Alessandro Cimatti, and Cristian Mattarei. Formal reliability analysis of redundancy architectures. *Formal Asp. Comput.*, 31(1):59–94, 2019.

[BCMG15] M. Bozzano, A. Cimatti, C. Mattarei, and A. Griggio. Efficient Anytime Techniques for Model-Based Safety Analysis. In *CAV*, pages 603–621, 2015.

[BCMT14] M. Bozzano, A. Cimatti, C. Mattarei, and S. Tonetta. Formal Safety Assessment via Contract-Based Design. In *ATVA*, volume 8837 of *LNCS*, pages 81–97. Springer, 2014.

[BCP+15] M. Bozzano, A. Cimatti, A. Fernandes Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, and S. Tonetta. Formal Design and Safety Analysis of AIR6110 Wheel Brake System. In *Proc. CAV*, volume 9206 of *LNCS*, pages 518–535, 2015.

[BCRZ99] A. Biere, E.M. Clarke, R. Raimi, and Y. Zhu. Verifiying Safety Properties of a Power PC Microprocessor Using Symbolic Model Checking without BDDs. In N. Halbwachs and D.A. Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1999.

[BCS02] P. Bieber, C. Castel, and C. Seguin. Combination of Fault Tree Analysis and Model Checking for Safety Assessment of Complex System. In F. Grandoni and P. Thévenod-Fosse, editors, *Dependable Computing - EDCC-4, 4th European Dependable Computing Conference, Toulouse, France, October 23-25, 2002, Proceedings*, volume 2485 of *Lecture Notes in Computer Science*, pages 19–31. Springer, 2002.

[BCT07] M. Bozzano, A. Cimatti, and F. Tapparo. Symbolic Fault Tree Analysis for Reactive Systems. In K.S. Namjoshi, T. Yoneda, T. Higashino, and Y. Okamura, editors, *Automated Technology for Verification and Analysis, 5th International Symposium, ATVA 2007, Tokyo, Japan, October 22-25, 2007, Proceedings*, volume 4762 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 2007.

[Bou17] Marc Bouissou. A benchmark on reliability of complex discrete systems: Emergency power supply of a nuclear power plant. In Holger Hermanns and Peter Höfner, editors, *Proceedings 2nd Workshop on Models for Formal Analysis of Real Systems, MARS@ETAPS 2017, Uppsala, Sweden, 29th April 2017.*, volume 244 of *EPTCS*, pages 200–216, 2017.

[Boz17] M. Bozzano. Causality and Temporal Dependencies in the Design of Fault Management System. *EPTCS*, 259:39–46, 2017.

[Bra11] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In Ranjit Jhala and David A. Schmidt, editors, *VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.

[Bry92] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[BV07] M. Bozzano and A. Villafiorita. The FSAP/NuSMV-SA Safety Analysis Platform. *STTT*, 9(1):5–24, 2007.

[BV10] M. Bozzano and A. Villafiorita. *Design and Safety Assessment of Critical Systems*. CRC Press (Taylor and Francis), an Auerbach Book, 2010.

[BVÅ03] M. Bozzano, A. Villafiorita, and O. Åkerlund et al. ESACS: An Integrated Methodology for Design and Safety Analysis of Complex Systems. In *Proc. European Safety and Reliability Conference (ESREL 2003)*, pages 237–245. Balkema Publisher, 2003.

[CCD+14] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv Symbolic Model Checker. In *CAV*, pages 334–342, 2014.

[CCGR00] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: a new symbolic model checker. *Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[CCM+18] Roberto Cavada, Alessandro Cimatti, Sergio Mover, Mirko Sessa, Giuseppe Cadavero, and Giuseppe Scaglione. Analysis of Relay Interlocking Systems via SMT-based Model Checking of Switched Multi-Domain Kirchhoff Networks. In Nikolaj Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE, 2018.

[CGMT13] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. Parameter synthesis with IC3. In *Proceedings of FMCAD*, pages 165–168. IEEE, 2013.

[CMS17] Alessandro Cimatti, Sergio Mover, and Mirko Sessa. SMT-based analysis of switching multi-domain linear Kirchhoff networks. In Daryl Stewart and Georg Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 188–195. IEEE, 2017.

[CPC03] A. Cimatti, C. Pecheur, and R. Cavada. Formal Verification of Diagnosability via Symbolic Model Checking. In *Proc. IJCAI*, pages 363–369, 2003.

[CRST12] A. Cimatti, M. Roveri, A. Susi, and S. Tonetta. Validation of requirements for hybrid systems: A formal approach. *ACM Transactions on Software Engineering and Methodology*, 21(4):22, 2012.

[DÅ04] J. Deneux and O. Åkerlund. A Common Framework for Design and Safety Analyses using Formal Methods. In *Conference on Probabilistic Safety Assessment and Management (PSAM7/ESREL'04)*, 2004.

[ea10] N. Hadjsaid et. al. Novel architectures and operation modes of distribution network to increase dg integration. In *IEEE conference, General Meeting 2010, Minneapolis, USA*. IEEE, 2010.
[ECS] European Cooperation on Space Standardization. http://www.ecss.nl.
[EDF19] Results of the electrical system dependability benchmark launched by EDF in 2017: demonstration of tools and models, 2019. https://www.imdr.eu/offres/gestion/events_818_46557_non-2229/results-of-the-electrical-system-dependability-benchmark-launched-by-edf-in-2017-demonstration-of-
html.
[ELMV11] J. Ezekiel, A. Lomuscio, L. Molnar, and S.M. Veres. Verifying Fault Tolerance and Self-Diagnosability of an Autonomous Underwater Vehicle. In *IJCAI*, pages 1659–1664, 2011.
[Eur07] European Space Agency. ESTEC ITT AO/1-5458/07NL/JD "System-Software Co-Engineering: Performance and Verification", 2007.
[Eur10] European Space Agency. ESTEC ITT AO/1-6570/10/NL/LvH "Dependability Design Approach for Critical Flight Software", 2010.
[Eur11] European Space Agency. ESTEC ITT AO/1-6992/11/NL/JK "FDIR Development and Verification & Validation Process", 2011.
[Eur13a] European Space Agency. ESTEC ITT AO/1-7263/12/NL/AK "Hardware-Software Dependability for Launchers", 2013.
[Eur13b] European Space Agency. ESTEC ITT AO/1-7785/14/NL/MH "Catalogue of System and Software Properties", 2013.
[GFB+14] A. Guiotto, R. De Ferluc, M. Bozzano, A. Cimatti, M. Gario, and Y.Yushtein. Fame process: A dedicated development and V&V process for FDIR. In *Proc. DASIA*, 2014.
[GVDM04] P. Gammie and R. Van Der Meyden. Mck: Model checking the logic of knowledge. In *CAV*, pages 256–259. Springer, 2004.
[Hua13] X. Huang. Diagnosability in concurrent probabilistic systems. In *AAMAS*, pages 853–860, 2013.
[HV89] J.Y. Halpern and M.Y Vardi. The complexity of Reasoning About Knowledge and Time. Lower Bounds. *Journal of Computer and System Sciences*, 38(1):195–237, 1989.
[JMWH05] A. Joshi, S.P. Miller, M. Whalen, and M.P.E. Heimdahl. A Proposal for Model-Based Safety Analysis. In *Proc. AIAA / IEEE Digital Avionics Systems Conference (DASC)*, 2005.
[KIHD08] F. Katiraei, R. Iravani, N. Hatziargyriou, and A. Dimeas. Microgrids management. *IEEE Power Energy Magazine*, 6(3):54–65, 2008.
[LMS02] F. Laroussinie, N. Markey, and P. Schnoebelen. Temporal logic with forgettable past. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 383–392. IEEE Computer Society, 2002.
[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In Rohit Parikh, editor, *Logics of Programs*, volume 193, pages 196–218. Springer Berlin Heidelberg, 1985.
[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.
[MIS15] MISSA. The MISSA Project, Last retrieved on January 28, 2015. http://www.missa-fp7.eu.
[NuS19] NuSMV web page, 2019. https://nusmv.fbk.eu.
[nuX19] nuXmv web page, 2019. https://nuxmv.fbk.eu.
[ocr19] ocra web page, 2019. https://ocra.fbk.eu.
[Pap00] Y. Papadopoulos. *Safety-Directed System Monitoring Using Safety Cases*. PhD thesis, Department of Computer Science, University of York, 2000. Tech. Rep. YCST-2000-08.
[PBB+13] T. Prosvirnova, M. Batteux, P.-A. Brameret, A. Cherfi, T. Friedlhuber, J.-M. Roussel, and A. Rauzy. The AltaRica 3.0 project for Model-Based Safety Assessment. In *DCDS*, 2013.
[PCV+06] T. Peikenkamp, A. Cavallo, L. Valacca, E. Böde, M. Pretzer, and E.M. Hahn. Towards a Unified Model-Based Safety Assessment. In *SAFECOMP*, pages 275–288, 2006.
[PM01] Y. Papadopoulos and M. Maruhn. Model-Based Synthesis of Fault Trees from Matlab-Simulink Models. In *Proc. Conference on Dependable Systems and Networks (DSN 2001)*, pages 77–82, 2001.
[Pnu77] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.
[Ray18] D.P. Raymer. *Aircraft Design: A Conceptual Approach*. American Institute of Aeronautics, 2018. Sixth Edition.
[SAE96] SAE. ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996.
[Sch04] A. Schumann. Diagnosis of discrete-event systems using binary decision diagrams. *Workshop on Principles of Diagnosis (DX'04)*, pages 197–202, 2004.
[SSL+96] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. C. Teneketzis. Failure diagnosis using discrete-event models. *IEEE Transactions on Control Systems Technology*, 4(2):105–124, 1996.
[Var01] M.Y. Vardi. Branching vs. Linear Time: Final Showdown. In T. Margaria and W. Yi, editors, *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 1–22. Springer, 2001.
[VSD+02] W.E. Vesely, M. Stamatelatos, J. Dugan, J. Fragola, J. Minarick III, and J. Railsback. Fault Tree Handbook with Aerospace Applications, 2002. Prepared for NASA Office of Safety and Mission Assurance, NASA Headquarters, Washington, DC 20546.
[xSA19] xSAP web page, 2019. https://xsap.fbk.eu.

A. Appendix: Analyzing the TMG Model Using XSAP

The analysis of the TMG model, documented in this paper, in based on XSAP release 1.3.0, which can be downloaded at https://xsap.fbk.eu.

We report below the command lines to be used for reproducing the results in the paper. If not otherwise stated, we use the python interface provided by the tool, in a Linux environment, however the same results can be replicated using XSAP via its interactive shell. Details about XSAP scripts and commands can be found in the user manual, distributed with the tool.

Section 5.3

We assume to run the commands from the folder examples/fe/triple_modular_generator.

```
$> cd examples/fe/triple_modular_generator
```

Model Extension

```
$> python ../../../scripts/extend_model.py -v SC_TMG.smv SC_TMG_prob.fei
```

Similarly for other fault extension instructions (suffix .fei) files.

Section 6.1

We assume to run the commands from the folder examples/fe/triple_modular_generator.

```
$> cd examples/fe/triple_modular_generator
```

Functional Verification Functional verification can be performed using the NUXMV model checker. The NUXMV commands are exposed in the XSAP shell.

```
$> ../../../bin/xSAP -int -dynamic out/extended_SC_TMG.smv
$> go
$> go_bmc
$> check_invar_ic3
$> check_ltlspec_ic3
$> check_ctlspec
$> show_property
```

Section 6.3

We assume to run the commands from the folder examples/fe/triple_modular_generator.

```
$> cd examples/fe/triple_modular_generator
```

Fault Tree Generation

```
$> python ../../../scripts/compute_ft.py -v --smv-file out/extended_SC_TMG.smv \
        --fms-file out/fms_SC_TMG.xml --prop-name R2_false --probability \
        --engine ic3 -b
```

The list of minimal cut sets and their statistics are listed in the log file out/xsap_compute_ft.out.

```
$> cat out/xsap_compute_ft.out
```

The fault tree can be visualized using the fault tree viewer.

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

The symbolic probability function can be generated by adding the option `--symbolic`.

Analysis with common causes and latent faults can be obtained by first extending the model with the desired fault extension instructions (suffix .fei) file, and then re-running fault tree generation.

FMEA Table Generation

```
$> python ../../../scripts/compute_fmea_table.py -v \
        --smv-file out/extended_SC_TMG.smv --fms-file out/fms_SC_TMG.xml \
        --prop-names R2_false -N 3
```

Similarly for other cardinalities.
The FMEA table is generated in file out/extended_SC_TMGfmea_table.txt.

```
$> cat out/extended_SC_TMGfmea_table.txt
```

Section 7.2

We assume to run the commands from the folder examples/FDI/diag.

```
$> cd examples/FDI/diag
```

Diagnosability Analysis

```
$> python ../../../scripts/check_diagnosability.py \
          --smv-file ../extended_SC_TMG_empty_controller.smv --asl-file G1.asl \
          --observables-file G1_observables.obs --engine ic3
```

Traces can be inspected using the trace viewer.

```
$> python ../../../scripts/view_trace.py \
          __xsap_prefix__extended_SC_TMG_empty_controller_trace1_A.xml
$> python ../../../scripts/view_trace.py \
          __xsap_prefix__extended_SC_TMG_empty_controller_trace1_B.xml
```

Similarly for other ASL specification files, namely G1_with_context.asl and G1_with_context_bounded.asl.

Observables Synthesis

```
$> python ../../../scripts/minimize_observables.py \
          --smv-file ../extended_SC_TMG_empty_controller.smv \
          --asl-file G1_with_context.asl --observables-file full_observables.obs
```

Diagnoser Synthesis

```
$> python ../../../scripts/synthesize_fd.py \
          --smv-file ../extended_SC_TMG_empty_controller.smv
          --asl-file G1_with_context_bounded.asl \
          --observables-file G1_state.obs --out-file G1_synthesized_model.smv
```

The output model is generated in file G1_synthesized_model.smv.
Completeness of the diagnoser can be verified as follows (similarly to functional verification).

```
$> ../../../bin/xSAP -int -dynamic G1_synthesized_model.smv
$> go_bmc
$> check_ltlspec_ic3 -p"(G F [0,5] CN.cmd_G1 = cmd_on) -> \
                        (G (SC.G1.Gen_StuckOff.mode != NOMINAL -> \
                         (F [0,5] __myfdir.myfd.Kalarm_G1)))"
```

Section 8

We assume to run the commands from the folder examples/fe/triple_modular_generator/parametric_models.
In the commands below, replace X with i for $i = 3, \ldots, 20$, to analyze the corresponding parametric model.

Model Extension

```
$> python ../../../../scripts/extend_model.py -v SC_TMG_X.smv ../SC_TMG_prob.fei
```

We provide below the commands to run fault tree analysis. We list the command for the PARAM algorithm, followed by the one for the BMC-PARAM algorithm.
The list of minimal cut sets and their statistics can be retrieved in the log file out/xsap_compute_ft.out.

Fault Tree Generation, Table 2

```
$> python ../../../../scripts/compute_ft.py -v \
          --smv-file out/extended_SC_TMG_X.smv --fms-file out/fms_SC_TMG_X.xml \
          --prop-name R2_false --probability --engine ic3 -b
$> python ../../../../scripts/compute_ft.py -v \
          --smv-file out/extended_SC_TMG_X.smv --fms-file out/fms_SC_TMG_X.xml \
          --prop-name R2_false --probability --engine bmc_ic3 -k 2 -b
```

Fault Tree Generation, Table 3

```
$> python ../../../../scripts/compute_ft.py -v \
          --smv-file out/extended_SC_TMG_X.smv --fms-file out/fms_SC_TMG_X.xml \
          --prop-text "(! SC.B1.is_powered)" --probability --engine ic3 -b
$> python ../../../../scripts/compute_ft.py -v \
          --smv-file out/extended_SC_TMG_X.smv --fms-file out/fms_SC_TMG_X.xml \
          --prop-text "(! SC.B1.is_powered)" --probability --engine bmc_ic3 -k 2 -b
```

Fault Tree Generation, Table 4

```
$> python ../../../../scripts/compute_ft.py -v \
          --smv-file out/extended_SC_TMG_X.smv --fms-file out/fms_SC_TMG_X.xml \
          --prop-name R2_false --probability --engine ic3 -b --faults-bound 4
$> python ../../../../scripts/compute_ft.py -v \
          --smv-file out/extended_SC_TMG_X.smv --fms-file out/fms_SC_TMG_X.xml \
          --prop-name R2_false --probability --engine bmc_ic3 -k 2 -b --faults-bound 4
```