Under consideration for publication in Formal Aspects of Computing

Formal reliability analysis of redundancy architectures

Marco Bozzano[1], Alessandro Cimatti[1], and Cristian Mattarei[2]

[1]Fondazione Bruno Kessler, Trento (Italy)
[2]Stanford University, Stanford, California (USA)

Abstract. Reliability is a fundamental property for critical systems. A thorough evaluation of the reliability is required by the certification procedures in various application domains, and it is important to support the exploration of the space of the design solutions.

In this paper we propose a new, fully automated approach to the reliability analysis of complex redundant architectures. Given an abstract description of the architecture, the approach automatically extracts a fault tree and a symbolic reliability function, i.e. a program mapping the probability of fault of the basic components to the probability that the overall architecture deviates from the expected behavior.

The proposed approach heavily relies on formal methods, by representing the architecture blocks as Uninterpreted Functions, and using the so-called *miter* construction to model the deviation from the nominal behavior. The extraction of all the deviation conditions is reduced to an AllSMT problem, and we extract the reliability function by traversing the Binary Decision Diagram corresponding to the quantified formula. Predicate abstraction is used to partition and speed up the computation.

The approach has been implemented leveraging formal tools for model checking and safety assessment. A thorough experimental evaluation demonstrates its generality and effectiveness of the proposed techniques.

Keywords: Redundancy Architectures; Triple Modular Redundancy (TMR); Reliability Analysis; Fault Tree Analysis (FTA); Satisfiability Modulo Theory (SMT); Equality and Uninterpreted Functions (EUF); Predicate Abstraction

1. Introduction

A key property in high-dependability, safety critical systems is the ability to continue to operate correctly even in presence of faults. This property, known as fault tolerance, can be achieved in many different ways (e.g., [KK07]). Among these, *redundancy* is one of the most used: components carrying out important functions are replicated, so that faulty ones can be identified and excluded upon reconfiguration, without compromising the overall functionality. An example of a widely adopted architectural pattern is Triple Modular Redundancy (TMR)[AS74, AL81, DR01, FM04, TIC$^+$05, JW10]. TMR consists in combining three replicated components, running in parallel, by means of a two-out-of-three majority voter. This approach guarantees that the generated output is correct even if one component

Correspondence and offprint requests to: Cristian Mattarei, Stanford University, 353 Serra Mall, Stanford, California, 94305. e-mail: mattarei@stanford.edu

fails, and has been used since the early days of computer-based systems, as in the 1969 Saturn Launch Vehicle Digital Computer [Cor64].

In order to design systems that are better equipped to deal with faults, many forms of analysis are required. At a qualitative level, it is important to guarantee that the system is able to tolerate a given number of faults, i.e. to operate correctly even when one or more basic components fail. At the quantitative level, it may be important to compute the reliability of the system, i.e. the probability of correct operation, as a function of the fault probability for the basic elements of the system.

Despite these practical needs, the analysis of architectures based on redundancy is a very heavy task, due to the lack of specific and automated techniques. Even recent results [HTK10] rely on a substantial amount of manual activity, carried out with "paper-and-pencil" techniques, and are limited by substantially simplifying hypotheses (e.g., that all the computing modules have the same failure probability).

In this paper, we propose a new approach to the reliability analysis of redundant architectures, which covers different types of TMR schemas. The user models the system under analysis using an architecture description language, where the components are equipped with sets of Boolean fault variables, which determine the behavior when one or more faults occur. Additionally, it is possible to specify the behavior of individual components as a function over the reals. The subsequent phases are completely automated. First, a model of the deviation of the system under analysis from its nominal behavior is built. Such a deviation can be seen as a *feared event*, also known as Top-Level Event (TLE). Then, the set of all fault configurations, also referred to as "cut sets" (CS), is computed [VSD$^+$02]. Intuitively, a CS is a set of faults under which the TLE occurs (i.e., the redundant architecture deviates from its nominal behavior). Once the cut sets are computed, they can be arranged into a fault tree, and they can be used to analyze the level of fault tolerance of the architecture: for example, if the TLE is only caused by cut sets containing at least two faults, it is possible to conclude that the system always tolerates one fault (i.e., there is no single point of failure). From the set of CS, it is also possible to extract a reliability function for the architecture under analysis, in form of a (program implementing a) mapping from the fault probability for the basic faults to the probability of the TLE. The reliability function can be evaluated to compute the probability of the TLE for different probabilities of the individual blocks, and to compare different choices, thus supporting the exploration of the design space.

The approach relies on the use of various formal techniques. We work in the framework of Satisfiability Modulo Theories (SMT) [BSST09], which is satisfiability with respect to some background mathematical theory. The notation $SMT(T)$ is used to indicate the dependency on the theory T. Basic blocks are represented in the theory of Equality and Uninterpreted Functions (EUF). Namely, an uninterpreted function is used to represent the behavior of a component as a generic function over the reals. Redundancy is modeled by reusing the same function symbols, combined with the circuitry representing the voting mechanism. The occurrence of faults is modeled with the introduction of Boolean fault variables. The model of the deviation is then an SMT formula with respect to the theory of EUF, written an $SMT(\text{EUF})$ formula. This formula is obtained with a so-called *miter* [Bra93] construction between two copies of the architecture model, where one of the copies, acting as a reference for the nominal behavior, is constrained not to fail. We leverage automated techniques from Model-Based Safety Analysis (MBSA) [JHMW06, BCT07, BCK$^+$14, BCP$^+$15] for the construction of the set of CSs. Within this framework, this computation reduces to an AllSMT problem [LNO06] on the miter formula, where the satisfying assignments to the fault variables correspond to fault configurations. The extraction of the reliability function is based on the analysis of the Binary Decision Diagram [Bry86] corresponding to the set of CSs.

Systems of realistic size may be associated with huge numbers of cut sets. Thus, the direct use of AllSMT [LNO06] for its computation turns out to be inefficient, due to the enumerative nature of the algorithm that ends up constructing the resulting formula in DNF.

We thus propose a new method for the compositional computation of the CSs. The key technical insight consists in reducing the problem to an equivalent Boolean problem by means of predicate abstraction [GS97]. We devised a two-steps method for the computation of the CSs for a given architecture. First, we combine the abstraction of the individual components under a suitable set of predicates, hence obtaining a purely Boolean model. Thus, we trade a huge AllSMT computation for several SMT-based quantifier eliminations, one per TMR type. Second, we compute the fault tree for such model using BDD-based projection techniques [BCT07]. We prove that the approach is sound, i.e., the fault trees computed on the abstract system are the same as the ones computed directly on the original, concrete system. This results in an algorithm where the computation is a highly structured combination of BDD-based quantifier eliminations. The compositional method is essential to our approach, since a flat analysis would lead to a combinatorial explosion among unrelated Boolean variables.

We implemented the approach by leveraging several existing tools for formal reasoning. The OCRA system [CDT13] is used to specify the system architecture. The NUXMV model checker [CCD$^+$14], which relies on the SMT solver MATHSAT [CGSS13], is used for predicate abstraction and quantifier elimination. The XSAP platform for model-

based safety assessment [BBC+16] provides the algorithms for the computation of CSs and supports the extraction of reliability functions. The tools support the process by providing the construction of the miter, and a comprehensive library of predefined redundancy patterns, including multiple TMR blocks.

The approach was experimentally evaluated on a number of examples and architectures of interest. The results demonstrates the following key properties. First, the method is very general: thanks to the expressive modeling language, it is possible to describe arbitrary redundancy architectures (e.g., [AS74, Yeh96, LJL07, HTK10, BCP+15, Mat16]). Second, the approach is fully automatic. The only requirement to the user is the modeling of the architecture using a set of guidelines. The output is a fault tree and a reliability function in form of an executable program, which can be used then for design space exploration. Third, the approach is highly scalable. The compositional method based on predicate abstraction proves to be dramatically superior to the concrete one, and allows us to analyze some classical architectures in a fraction of a second. Such a significant improvement, compared to classical techniques, enables the possibility to analyze larger systems, as well as evaluate a broader set of alternative architectures in the same amount of time.

This paper is organized as follows. In Section 2 we discuss some relevant related work. In Section 3 we present some logical background. In Section 4 we describe redundancy architectures. In Section 5 we formally define the problem of reliability analysis. In Section 6 we describe the overall flow of our approach. In Section 7 we present the application of predicate abstraction for cut sets computation. In Section 8 we describe the implementation and the experimental evaluation. In Section 9 we draw some conclusions and outline directions for future work.

2. Related work

Analysis of redundant architectures

Our work is close to the one presented in [HTK10], and extends it along three main directions: i) [HTK10] presents an ad-hoc algorithm that can analyze the reliability of computational chains based on Triple Modular Redundancy with one voter. The user is required to provide the conditional probability of failures, for each pair of input and output port. Our approach requires only a formal model of the architecture, and safety and reliability analyzes are completely automated; ii) our approach is much more general, and deals not only with linear structures but also with tree- and DAG-like structures, and with a much wider class of TMR blocks; iii) the method in [HTK10] requires the user to provide a level of discretization. Our technique, instead, is purely symbolic, and generates a precise, closed form of the reliability function, where no discretization is required.

Other techniques to analyze TMR-based architectures are based on Monte Carlo simulations. For instance, [LJL07] relies on this technique in order to evaluate chains of TMR shift registers. The simulation based approaches do not provide exhaustive evaluation of the system, and they require a behavioral definition of the modules.

The modeling technique that we propose in this paper may recall the Reliability Block Diagrams (RBD) [Čep11], which are used to describe functional dependencies between architectural components. However, our approach is strictly more expressive, since we are not limited to describe (generic) component dependencies but we can also provide interpreted behaviors, as well as more detailed dependencies. Our modeling framework is similar to the relational language introduced in [JS91]. In this work, we propose the use of SMT techniques to automate the reasoning part.

A configuration logic for describing parametric architectures is presented in [MBBS15]. Differently, in this paper we do not consider architectures with a parametric number of interacting components and adaptive interfaces, but we provide ways to analyze in full detail the reliability of static networks. Extending this approach to analyze networks with variable topology is the object of future work.

Formal analysis of redundant architectures

The techniques based on Markov Decision Process and Probabilistic Petri Nets [KKZ05, HKNP06, Tri02, CMT89, SIQW95] are widely used in industry for the quantitative evaluation and reliability analysis. However, such approaches cannot provide a uniform system modeling and completely automated process, and in fact, the link between the reliability evaluation and the qualitative safety analysis is performed manually.

The use of formal methods techniques to analyze redundancy architectures is rather limited. In [LQJ], the Communicating Sequential Processes (CSP) formalism is used to model and prove the correctness of a single TMR stage. The work is mostly manual, and does not include any quantitative analysis. [Jan97] provides a bi-simulation based technique to prove fault tolerance of systems modeled using calculus of communicating systems (CCS). In [ZLMR09], a module based on redundancy is designed within the formalism of timed automata, and analyzed using the Uppaal

model checker. This work focuses on the specific features of the design, and does not consider multi-staged architectures.

[LQJ] can aid the verification and reliability analysis in the limited case for TMR chains. However, such approaches cannot be generalized in order to cover a full set of architectural patterns.

Model based safety assessment

This work can be framed within the line of Model-Based Safety Assessment (MBSA) [JHMW06]. The perspective of MBSA is to represent the system by means of a formal model, and perform safety analysis (both for preliminary architecture and at system-level) using formal verification techniques. The integration of these techniques allows safety analysis to be more precise and more cost-effective. Such techniques must be able to verify functional correctness and assess system behavior in presence of faults [BVÅ$^+$03, ABB$^+$06, BV10, JH05, BS97].

At the core of model based safety assessment is the ability to exhaustively analyze the behaviors of dynamical systems. Traditionally, dynamical systems are modeled as finite state systems: their state can be represented by means of assignments to a specified set of variables [Hol97]. Techniques for Model-Based Safety Assessment include Fault Tree Analysis [VGRH81] and Failure Mode and Effects Analysis (FMEA), which can be performed automatically by reduction to symbolic model checking [BV10, BCK$^+$11, BV07, BCT07, BCL$^+$11, BCP$^+$15].

Engines

A key difference with respect to our approach is that the existing techniques for MBSA focus on the analysis of the *behavior* of dynamical systems, whereas our approach aims at evaluating characteristics of redundancy architectures, independently of components' behavior. More specifically, we rely on the calculus of Equality and Uninterpreted Functions (EUF) and we make use of Satisfiability Modulo Theory (SMT) techniques for their analysis [BGL$^+$00, FHT$^+$07]. Moreover, we extend this technique to a partitioned quantification that relies first on SMT and afterwards on a BDD-based [Bry92] approach.

The analyzes of SMT formulas based on EUF theory can be approached also by applying Ackermann expansion [LS04, BCF$^+$06], thus substituting the function symbols with a set of constraints. However, while such transformation preserves satisfiability, it does not guarantee equivalence, which is fundamental when dealing with minimal cut sets computation.

3. Background

We work in the setting of first order logic. Individual terms are either constants, individual variables, or the application of function symbols of arity n to n terms. Atoms are either Boolean variables, or the application of an n-ary relational symbol to n terms. Formulae are either the truth values $\top$ and $\bot$, atoms, the application of an n-ary Boolean connective (e.g. unary negation $\neg$, binary conjunction $\wedge$, disjunction $\vee$, implication $\rightarrow$, double implication $\leftrightarrow$), or the application of a quantifier (existential $\exists$, universal $\forall$) to a variable and a formula. We write $\phi[v/\psi]$ to denote the substitution of every occurrence of v in ϕ with ψ. We use a generalized notation for quantification and substitution to sets/vectors of variables. A literal is either an atom or its negation. A clause is a disjunctions of literals. A formula in conjunctive normal form is a conjunction of clauses.

Boolean formulae are quantifier-free formulae whose atoms are Boolean variables. Quantified Boolean Formulae (QBF) are formulae where atoms are Boolean variables. QBF are as expressive as Boolean formulae, based on the fact that $\exists P.\phi \equiv (\phi[P/\bot] \vee \phi[P/\top])$ and $\forall P.\phi \equiv (\phi[P/\bot] \wedge \phi[P/\top])$.

Binary Decision Diagrams [Bry86] are a canonical representation for Boolean and QBF formulae. They have been historically used in symbolic model checking to represent sets of states and to implement image computation by means of advanced quantification routines [McM07, RAB$^+$95]. The problem of satisfiability of Boolean formulae (SAT) has been heavily investigated, and efficient SAT solvers are available [SLM09]. Satisfiability Modulo Theory (SMT) [BSST09] is an extension of the SAT decision problem, where the formula is not pure Boolean, but it is expressed in some background theory, where some symbols are interpreted. Theories of interest include linear arithmetic over the reals (LRA) and over the integers (LIA), the theory of arrays, and the theory of bit vectors. In the rest of this paper we will focus primarily on the theory of Equality and Uninterpreted Functions (EUF), where function symbols can be declared, but have no specific property, except for the fact that they are functions, i.e., $(x = y) \rightarrow (f(x) = f(y))$. We write $\mathbb{B}$ for the Boolean type, and $\mathbb{D}$ for the type of uninterpreted objects.

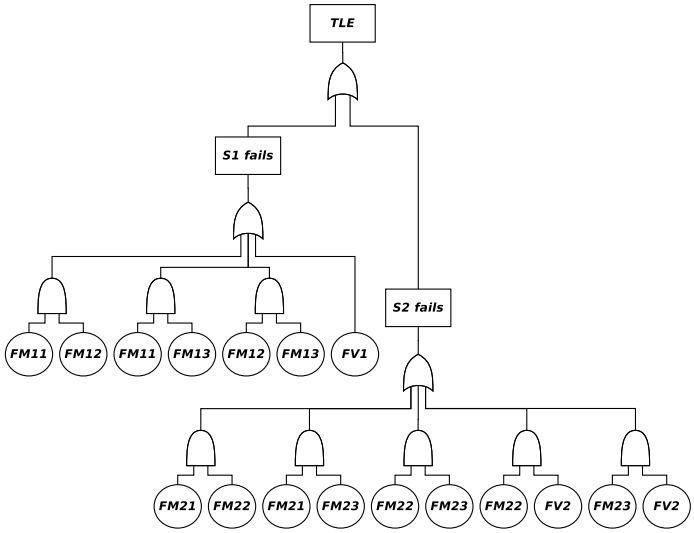


Fig. 1. Fault tree, two TMRs ([a,b] as in Figure 2)

In addition to satisfiability, useful functionalities for SMT are quantifier elimination and AllSMT [LNO06]. Quantifier elimination is the process of finding, for a given formula, an equivalent quantifier-free formula. AllSMT is a technique to enumerate all solutions to an SMT problem, and it can be also used as a quantifier elimination technique. The standard approach to AllSMT is a sequence of incremental SMT calls, where the satisfying assignments to the remaining variables are enumerated and blocked. We notice that AllSMT constructs a Disjunctive Normal Form (DNF) representation of the resulting formula.

Fault Tree Analysis (FTA) is a technique for reliability and safety analysis, based on the construction of a Fault Tree Diagram, or simply Fault Tree (FT) [VSD+02]. A FT is a representation of the possible scenarios that enable an undesirable configuration, also called Top Level Event (TLE), to be reached. A Fault Tree is characterized by four kinds of nodes (see example in Figure 1):

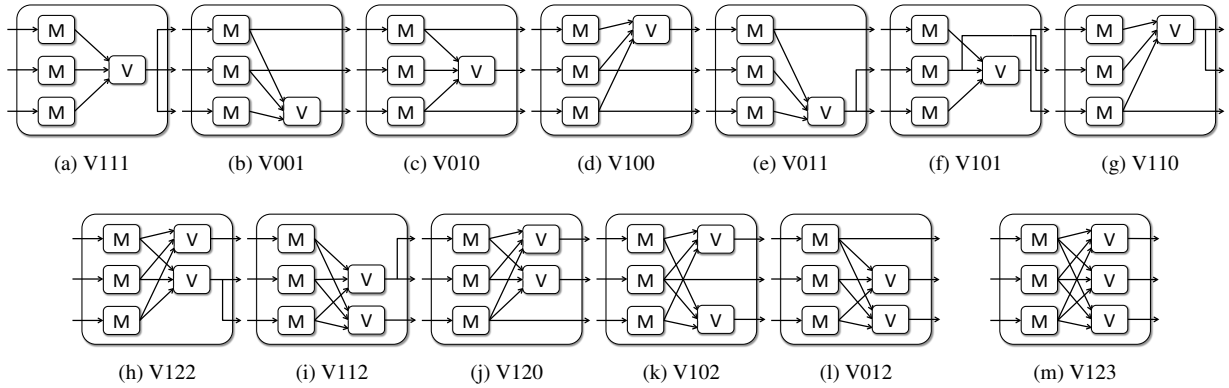
- basic faults (circles, name starts with "F"): they are the leaves of the tree and represent the faults of basic components, e.g., "the generator is broken" or "the switch is stuck at open";
- intermediate events (boxes "S1 fails" and "S2 fails"): they represent an hazardous condition reached by a subsystem;
- top level event (box "TLE"): represents an undesirable configuration, also known as feared event, which is reachable by the system;
- logic gates (ANDs and ORs gates) define the relation between the nodes of the tree.

A FT represents a collection of cut sets. A cut set is a set of faults that can cause the Top Level Event. It is possible to interpret a FT as a Boolean formula, whose satisfying assignments to the basic fault variables correspond to cut sets. Often FTA works under the monotonicity assumption, i.e. if the Top Level Event can be caused by the cut set c, then it can also be caused by any superset of c. A cut set is said to be minimal if none of its proper subsets is a cut set.

4. Redundancy architectures

Redundancy is a widely adopted solution when dealing with critical systems. The basic idea is to integrate in the system multiple copies of the same component, such that a single fault will not cause the failure of the entire system. There are many forms of redundancy. Some are based on (either cold, warm, or hot) standby, and reconfiguration, and are directed to increasing availability. In order to increase reliability, i.e. the overall likelihood of correct computation, we consider "active-active" architectures, where multiple copies of the same component are run in parallel, and the results are combined by means of voting.

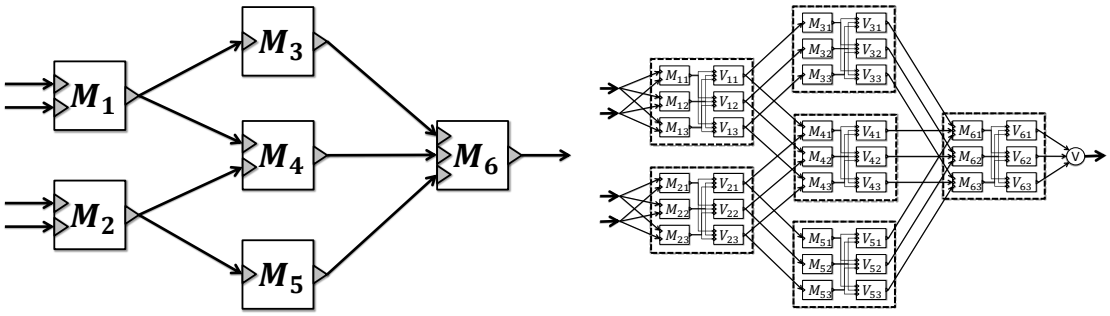
A typical example of redundancy schema is Triple Modular Redundancy (TMR). This design pattern is widely used for aircraft, nuclear reactor plants, railways, and electric supply systems [Yeh96, BV10, Mon93, lay]. A basic



(a) V111 (b) V001 (c) V010 (d) V100 (e) V011 (f) V101 (g) V110

(h) V122 (i) V112 (j) V120 (k) V102 (l) V012 (m) V123

Fig. 2. Triple Modular Redundancy (1, 2 and 3 voters per stage)



(a) Reference System (b) Redundant System

Fig. 3. Network of combinatorial components [AS74]

TMR block is composed by three replicated modules, the results of which are combined by a voter component. If all the components are in agreement, the voter returns one of the values. If only two of the components agree, the value computed by the "majority" is returned. For example, the voter schema used in [Yeh96] returns the median of the input values. It is easy to see that a TMR schema is able to tolerate a single fault. Other forms of redundancy, such as Dual Modular Redundancy, are also applied in practice [BLBM07]. In the following, without loss of generality, we focus on TMR-based architectures.

In order to compose multiple TMR blocks into more complex architectures, triplicated inputs and outputs are considered. TMR blocks may have up to three voters, and different connections between them. The various combinations are shown in Figure 2. For the sake of simplicity, we depict unary computing modules within TMR blocks.

In fact, computing modules of greater arity are also possible. The architecture from [AS74], depicted in Figure 3, will be used as a reference example in the rest of this paper. We notice that in addition to the unary modules M_3 and M_5 we have binary modules (M_1, M_2 and M_4) and a ternary module (M_6). On the left-hand side there is a description of the data flow being computed, and on the right a representation of the redundancy architecture (based on the V123 TMR block of Figure 2).

5. Reliability analysis: the problem

The problem tackled in this paper is reliability analysis, i.e. to evaluate the probability of failure of a redundant architecture. We are also interested in explaining the overall reliability in terms of the reliability of individual components. This can be done both in a qualitative way, namely comparing the cut sets / fault tree for different architectures, and in a quantitative way, namely comparing the probabilities. For the time being, we assume that the failure probabilities

of the components and of the architecture, are expressed relative to a given time interval, e.g. the mission time. Our framework is general, and can accommodate alternative approaches, e.g. reliability as a function of time.

Computing modules and voters are subject to faults, with frequency that is modeled in form of probability values. In the case of a fly-by-wire architecture as the Primary Flight Computer (PFC) described in [Yeh96], failure may mean that the actuators do not receive correct control outputs from the pilot. The ability to quantify the failure probability is required, for example, to demonstrate that a given architecture does obey the safety requirements (e.g., the overall failure probability is below a certain threshold). Reliability analysis also plays a fundamental role in design space exploration, when different architectural solutions are compared, based on parameters such as costs, weight, and design complexity. This supports the choice among different architectures that belong to the Pareto's frontier, and are locally optimal and incomparable solutions.

The computation of reliability is based on probability theory. Intuitively, we must accumulate all the possible configurations of basic faults under which the output of the architecture deviates from the expected one.

Single module

Let us consider first a single module. Assume the probability of failure for a computing element M is F_M, and for a voter V is F_V. Then, the probability of failure (producing a wrong output) for the TMR in Figure 2a (V111), assuming that the faults of the three computing elements and of the voter are independent, is symbolically expressed as

$$F_V + (1 - F_V) * ((\binom{3}{2} * (F_M)^2 * (1 - F_M)) + (F_M)^3)$$

which is equivalent to

$$F_V + (1 - F_V) * (3 * (F_M)^2 - 2 * (F_M)^3).$$

The formula combines the case where the voter fails, which is sufficient per se to corrupt the output, and the case where the corruption is the result of at least two computing elements failing.

Once a symbolic reliability function is available, interesting considerations can be drawn. For instance, a TMR approach is not always able to increase the overall reliability of the system, because this depends on the specific setting where each component implementation is going to operate. More specifically, the reliability analysis of different redundant approaches has to take into account the probability of failure of each single module and of the voter.

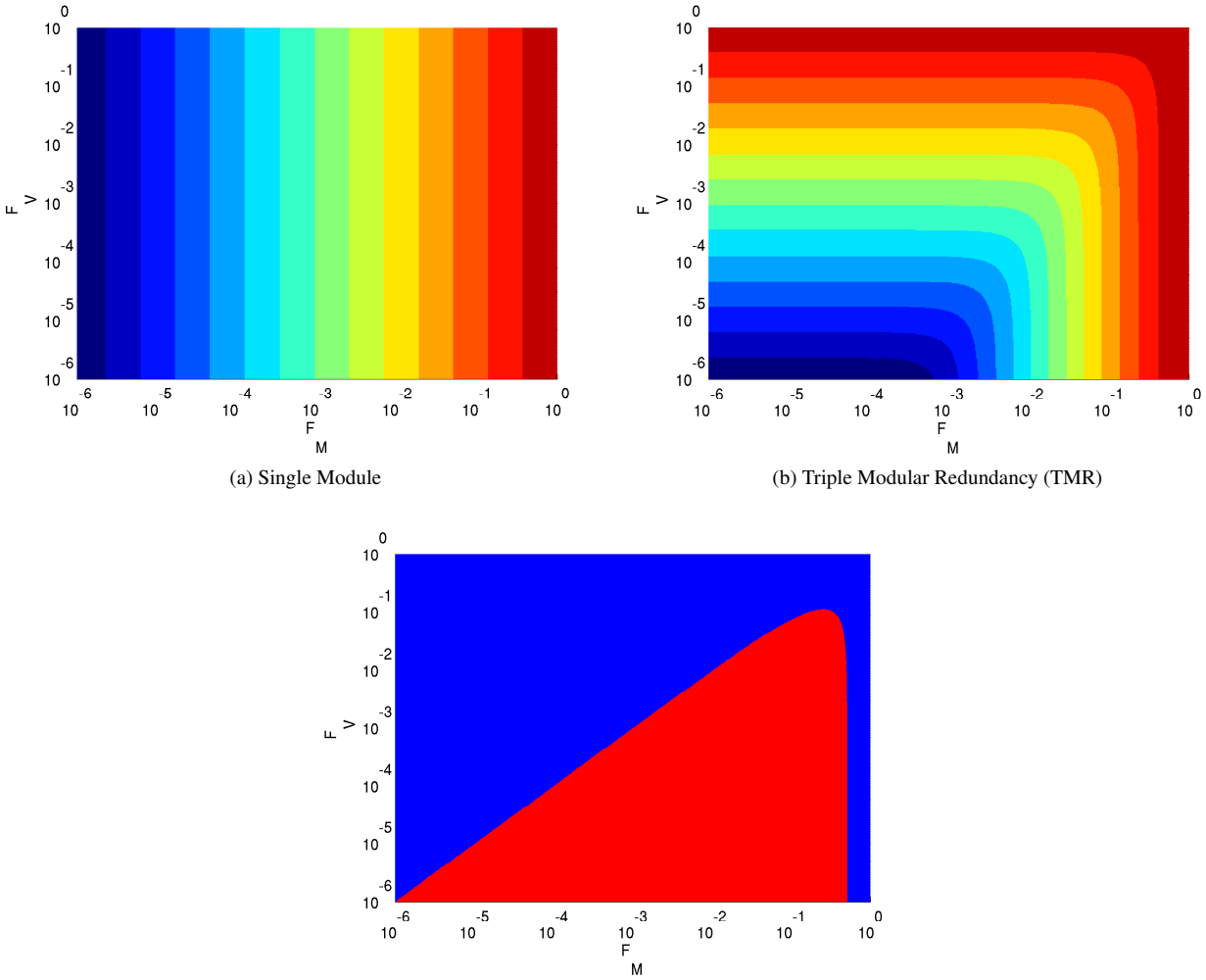
The 2-dimensional plots in Figure 4 show how a single module (Figure 4a) and a TMR approach (Figure 4b) deliver different probability of failure, by varying the reliability of modules and voters. The blue area represents a low probability of failure, while the red one a high probability of failure. Moreover, Figure 4c shows when a TMR approach delivers higher reliability (the red area) compared to a single module (the blue area), by varying the probability of failure of computational modules (F_M) and voters (F_V). This analysis shows that a TMR approach is better when the voter is more reliable than the computational module. This is, in general, a fair assumption because voter implementations are likely simpler than computational module ones, but there might be some cases where this assumption does not hold.

Figure 1 shows the generated fault tree, for a chain with a TMR of type V111 (Figure 2a) followed by a TMR of type V001 (Figure 2b). In this case, the TLE represents the inequality between TMRs and perfect modules. The intermediate event "S1 fails" specifies that at least 2 outputs of stage 1 diverge from the nominal value, whereas "S2 fails" represents, respectively, the same condition for stage 2.

Linear TMR architectures

A real scenario can be composed of hundreds of different computational modules, and the evaluation of such architectures needs to be performed via automated techniques.

Consider, for example, a chain of TMR modules of the form $M_1 \triangleright M_2 \triangleright \ldots \triangleright M_n$, as adopted, for instance, in [Cor64]. The reliability function is non-trivial, given that it needs to consider also different implementations of voters and modules, thus with a multiplicity of fault probabilities. The work in [HTK10] analyzes the properties of chains of TMRs of different patterns (e.g., the 1 voter cases shown in Figure 2). Assume now that the problem is to devise the best displacement of TMRs, given the existence of constraints limiting the number of voters (e.g., 5 voters overall). The outcome of the analysis is presented in Figure 5, under the hypothesis that the computing modules have the same fault probability (and also the voters): the X axis represents the probability of failure of the voter, Y axis the



(a) Single Module

(b) Triple Modular Redundancy (TMR)

(c) Single Module (Blue - upper part) vs. TMR (Red - lower part)

Fig. 4. Single Module and Triple Modular Redundancy comparison

probability of failure of each module. For each point, the color represents the architectural pattern that guarantees the highest system reliability for a pair of X, Y values.

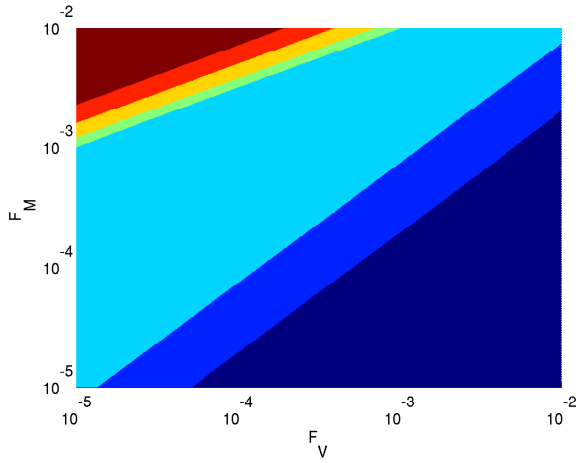
The technique proposed in [HTK10] is subject to several limitations: the user is requested to define the function that describes the reliability of each possible TMR; such results are composed by means of an automated technique that instantiates the functions on a set of possible numerical values of failure probability. Furthermore, the framework requires uniformity: a chain of the same TMR pattern, with the same fault probability.

Tree- and DAG-like architectures

Finally, there are architectures as the one in Figure 3, from [AS74], which contains computing modules of multiple arity, and multiple fan out.

6. Formal reliability analysis of redundancy architectures

We now present a formal, automated approach for the production of the reliability function for a given architecture. The approach is based on the following phases (detailed in the rest of the Section):



(a) Best architectures

Color	Array of configuration
(a) blue (bottom-right)	[b,b,b,b,b,b,b,b]
(b) blueberry	[b,b,c,b,b,b,c,b]
(c) lightblue	[b,c,d,b,c,d,b,c]
(d) green	[d,c,b,a,d,c,b,a]
(e) yellow	[d,a,d,a,d,a,d,a]
(f) red	[a,a,d,a,a,a,d,a]
(g) brown (top-left)	[a,a,a,a,a,a,a,a]

(b) Mapping colors to architecture

Fig. 5. Reliability comparison: TMR chains with 1 voter

Architecture modeling The architecture is modeled using uninterpreted function to express computing modules, and Boolean logic to express the connections between them. Uninterpreted functions allow us to express the functional properties of the behavior of individual components. At the same time, since the functions are uninterpreted, we can concentrate on the features of the redundant architecture, abstracting from the specific module implementations.

Miter composition First, we model a selective switch that allows us to enable or disable the possibility to have faulty behaviors on the entire architecture. Then, we compare two architecture instances where one can fail and the other cannot fail. The first one represents the architecture under analysis and the latter the reference one, while the miter composition is designed to provide them the same input and evaluating under which conditions they provide a different result.

Cut sets computation We construct the set of all fault configurations that are sufficient for the architecture under analysis to provide a wrong result. These configurations can be arranged into a fault tree representation. The analysis is performed by relying on the minimal cut sets computation on the miter composition.

Reliability Function extraction This phase constructs the BDD-based representation of the set of cut sets [Rau93, Rau01], and traverses it to generate the reliability function in closed form.

6.1. Architecture modeling

Figure 6 shows a graphical representation of a TMR with single voter. Basic computing elements are modeled as combinatorial, i.e. stateless components. Each module has two separate behaviors: nominal (M_N), and faulty (M_F). Those behaviors are described by relying on uninterpreted functions. The coherence between each nominal behavior is guaranteed by the fact that all modules, except voters, are sharing the same uninterpreted function (green arrows in Figure 6, labelled as "nominal_behavior"). The behavior under fault is assumed not to be constrained, and it is also represented as an uninterpreted function. Since the modules are not required to fail in the same way, each function can be local to each module. The voter V_N has a well defined implementation, and it does not need to be modeled with an uninterpreted function because it is interpreted and well defined. The outputs of each pair M_N/M_F and V_N/V_F are given to a multiplexer, which selects the right signal according with the fault event (represented with the red arrows). The input to each multiplexer can be masked with a *can_fail* signal in order to enable/disable the faulty behavior.

The formal model that describes the setting shown in Figure 6 is defined using the SMV language extended with the support for uninterpreted functions. Figure 7 presents the definition of the extended module. More in detail, the module receives three parameters: `input`, representing the input value of the computation (of type `real`); the Boolean parameter `can_fail`, which enables the component to have internal failures; and the function `nominal_behavior`, modeling the computation in the nominal case. The modeling approach relies on `real` data types in order to describe infinite domain values, referred later in this paper also as $\mathbb{D}$ type. Within the definition of the extended component we have: the variable `failure` that keeps track of the current behavior (nominal or faulty), the definition of the

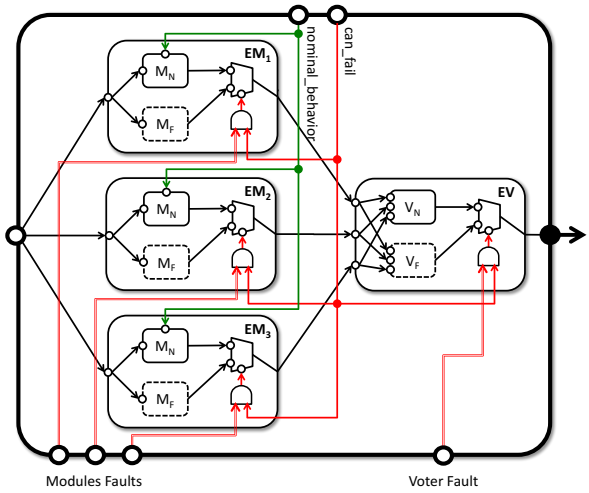


Fig. 6. TMR component with EUF

```
1   MODULE MODULE_1_INPUT(input, can_fail, nominal_behavior)
    VAR
      local_failure : boolean;
5
    FUN
      faulty_behavior : real -> real;
    DEFINE
10    failure := local_failure & can_fail;
    DEFINE
      output :=
        case
15        !failure : nominal_behavior(input);
          TRUE     : faulty_behavior(input);
        esac;
```

Fig. 7. An example of extended module (SMV language)

`faulty_behavior`, and the multiplexer (line *13* in Figure 7) that implements the switching between nominal and faulty behavior.

Figure 8 presents the definition of the extended voter. This component receives four parameters: the tree input values `input_1`, `input_2` and `input_3`, of type `real`, and the Boolean parameter `can_fail`, which enables the component to have internal failures. The definition of the extended voter is composed of: the variable `local_failure` that keeps track of the current behavior (nominal or faulty), the definition of the expected behavior of the voter (line *10*), the definition of the `faulty_behavior`, and the multiplexer (line *21* in Figure 7) that implements the switching between nominal and faulty behavior. The masking with the `can_fail` signal is represented in line *18*.

The EUF-based modeling approach allows for the description of complex computational networks, where each component can be extended with a redundancy pattern. More specifically, this modeling approach allows for extending any computational network, represented as a Directed Acyclic Graph (DAG), with any of the redundancy patterns examples listed in Figure 2.

6.2. Miter composition

The idea at the basis of the analysis of redundant architectures is to evaluate which component failures may affect the possibility to provide the correct value. In order to do this, we need to compare the faulty system, modeled as described in the previous section, with a reference architecture that always operates as expected e.g., with no faulty behavior.

An architecture composed of a set of modules with `nominal_behavior` and `can_fail` as parameters (as described in Figure 6) gives us the possibility to describe both reference and faulty systems. In fact, the reference architecture is instantiated by providing `FALSE` as `can_fail` parameter to all components, while it will be `TRUE`

```
MODULE VOTER_1_INPUT(input_1, input_2, input_3, can_fail)
VAR
  local_failure : boolean;

FUN
  faulty_behavior : real * real * real -> real;
DEFINE
  voted_output := case
                    input_1 = input_2 : input_1;
                    input_1 = input_3 : input_1;
                    input_2 = input_3 : input_2;
                    TRUE              : input_3;
                  esac;
DEFINE
  failure := local_failure & can_fail;
DEFINE
  output :=
    case
      !failure : voted_output;
      TRUE     : faulty_behavior(input_1, input_2, input_3);
    esac;
```

Fig. 8. An example of extended voter module (SMV language)

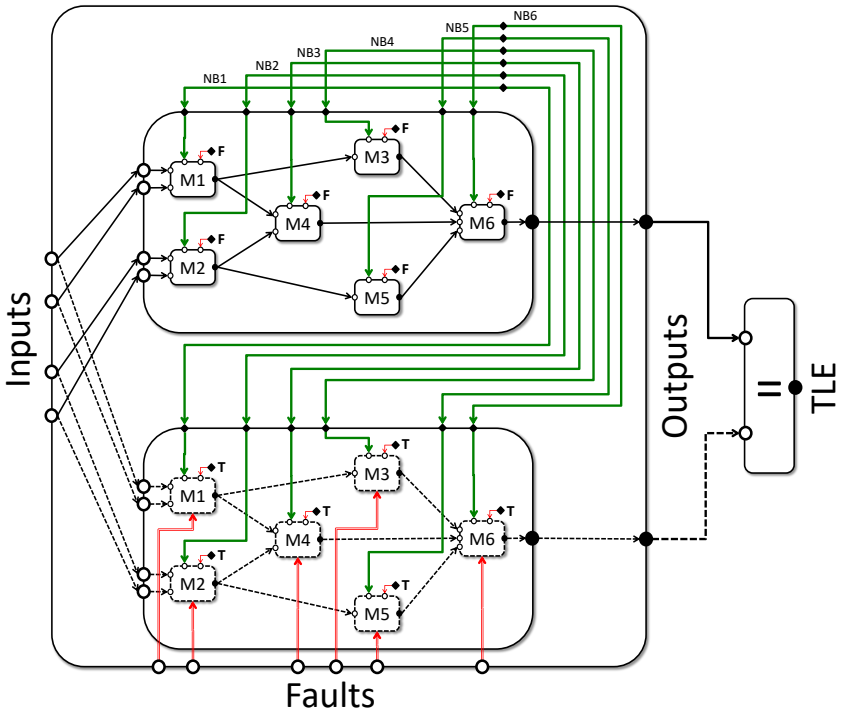


Fig. 9. Miter composition.

in case of a faulty description. Moreover, all `nominal_behaviors` will be shared between the two instances, in order to guarantee a coherence when no failures have occurred. This system composition, as shown in Figure 9, is called *miter* [Bra93]. An important aspect when comparing two architectures is to provide them the same inputs, and evaluate the difference in the outputs. A common analysis performed on the miter construction is to evaluate whether the reference and faulty outputs are equal. Another possible analysis could be to consider an output to be wrong only when it differs with respect to the expected one by a predefined threshold.

The miter construction comes with a distinguished signal TLE. If the model has a single output, then TLE is the deviation between the two copies, i.e. $o \neq o'$. If there are multiple outputs then the TLE is the disjunction of all the

output deviations, i.e.

$$\textsc{Tle} \doteq \bigvee_{o \in \vec{O}} (o \neq o')$$

6.3. Minimal Cut Sets computation

The miter composition, as shown in Figure 9, is evaluated in order to generate the set of conditions that may cause the two systems to provide different outputs in presence of the same inputs. More specifically, we are interested in the set of assignments to the fault variables, referred to as fault configurations, which are sufficient to provide a wrong result.

We require the reference architecture to be deterministic when the fault variables are set to false. Notice that without this hypothesis, two perfectly legal behaviors could trigger the Top Level Event, thus resulting in an empty (and meaningless) fault configuration.

Consider the TMR example in Figure 6. We call the three extended modules EM_1, EM_2 and EM_3. Then, the set of fault configurations, also called cut sets, which are sufficient to produce a wrong output are:

1. $\{EV\}$: every output of each module EM_i is evaluated by the voter EV, and this implies that a failure of such module is sufficient to cause a wrong behavior of the entire system.
2. $\{EM_1, EM_2\}$: the output of the voter EV is in accordance with the majority of the received inputs, and a wrong result produced by more than two modules is then propagated to the voter's output.
3. $\{EM_1, EM_3\}$: similar condition as $\{EM_1, EM_2\}$, but with EM_1 and EM_3.
4. $\{EM_2, EM_3\}$: similar condition as $\{EM_1, EM_2\}$, but with EM_2 and EM_3.

Every cut set can be represented, via a propositional formula, as a conjunction of component's faults, and the whole set of configurations as a disjunction of cut sets. According to these conditions, the resulting formula for the example in Figure 6 is

$$\mathrm{F}_{EV} \vee (\mathrm{F}_{EM_1} \wedge \mathrm{F}_{EM_2}) \vee (\mathrm{F}_{EM_1} \wedge \mathrm{F}_{EM_3}) \vee (\mathrm{F}_{EM_2} \wedge \mathrm{F}_{EM_3}) \tag{1}$$

In a general case, defining the miter composition as an SMT formula π over input ports $\vec{I}$, output ports $\vec{O}$, fault variables $\vec{F}$, and Top Level Event, the formula representing the set of cut sets is obtained as

$$\exists \vec{I}, \vec{O}. \left(\pi(\vec{I}, \vec{O}, \vec{F}) \wedge \textsc{Tle}(\vec{O}) \right) \tag{2}$$

Intuitively, this formula represents the set of assignments to $\vec{F}$ such that there exists an assignment to $\vec{I}$ that allows the two architectures to provide different output values.

The problem of extracting the cut sets is naturally encoded as an AllSMT for the theory of EUF [LNO06]. The AllSMT problem is a specific form of quantifier elimination: all the theory variables are eliminated, and only Boolean variables remain. In this case, the variables being quantified out are the input and output ports $\vec{I}$ and $\vec{O}$, while the remaining variables are the $\vec{F}$ variables. Intuitively, the AllSMT proceeds by finding a satisfying assignment, extracting the assignment to the remaining variables, transforming it into a blocking clause, and iterating until a fix point is reached.

We notice that the problem enjoys a monotonicity property: if an assignment μ to the fault variables can cause the TLE, so can all the assignments μ' where one or more $\bot$ values are replaced with $\top$, i.e. one or more components are turned from not faulty to faulty. Intuitively, if a set of faults can cause the Top Level Event, so can all its super-sets. This follows from the fact that in case of fault the behavior of the components is completely unconstrained: thus, in case of μ', it is still possible for a faulty component to exhibit the same behavior that was associated by μ.

This property can be exploited in the AllSMT computation. This is done in two ways. First, the splitting heuristics are modified so that the fault variables are given negative polarities, hence models with "less faults" are privileged. Second, during the AllSMT iteration, the model is generalized by removing all the negated fault literals before blocking.

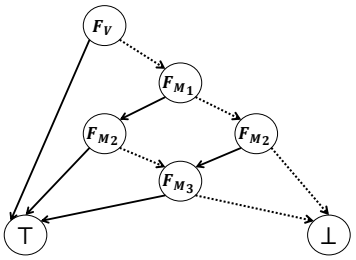


Fig. 10. BDD of the formula $\mathrm{F}_V \vee (\mathrm{F}_{M_1} \wedge \mathrm{F}_{M_2}) \vee (\mathrm{F}_{M_1} \wedge \mathrm{F}_{M_3}) \vee (\mathrm{F}_{M_2} \wedge \mathrm{F}_{M_3})$

6.4. Reliability function extraction

The formula representing the cut sets can be converted into a BDD-based representation. From this, we can automatically generate the reliability function, in a closed form fashion, of the entire system. We remark that the generation of the cut sets is a necessary step in the context of this work, since the models under analysis contain real-valued variables, hence BDD-based techniques cannot be directly applied on the models themselves.

The advantage of this approach consists in providing a result that is independent from the specific fault probability of the individual components. More specifically, the closed form of the reliability function represents the characteristic of the redundant pattern and is independent of the possible values of the failure probability. Moreover, the generation of the reliability function can be performed once and for all, and then computed on-demand on every instance of the problem.

Given the propositional formula representing failure configurations of a redundant system, and assuming independence between events, it is possible to extract the closed form of the reliability function by analyzing its BDD representation. If the Boolean BDD variables are intended to represent the occurrence of independent events, and every fault variable q is associated with a probability of occurrence F_q, then it is possible to associate a probability to the overall truth of the formula represented by the BDD. This is done recursively as follows, where n is a BDD node and n_1, n_2 its sub-nodes:

$$\textsc{BddProb}(n) = \begin{cases} 1 & \text{if } n = \top \\ 0 & \text{if } n = \bot \\ \mathrm{F}_q * \textsc{BddProb}(n_1) + & \\ \quad (1 - \mathrm{F}_q) * \textsc{BddProb}(n_2) & \text{if } n = ITE(q, n_1, n_2) \end{cases} \tag{3}$$

Equation 3 provides an intuitive characterization of the algorithm that extracts the reliability function from the BDD representation. This method essentially performs a depth-first-search visit over the BDD structure, considering all paths that lead to the $\top$ node. In the base cases, the $\top$ node results in a probability of 1 (regardless of possible variables assignments), and a $\bot$ node yields a probability of 0, as the corresponding assignment does not cause the TLE. The step case in Equation 3 expresses the ITE concept from the theory of probability, assuming that events are independent, with the positive occurrence of the variable q represented by F_q and, respectively, its negative occurrence with $(1 - \mathrm{F}_q)$.

Figure 10 shows the BDD representation of the cut sets produced by the example in Figure 6. The application of the approach described by Equation 3 to this example is presented by Equation 4, where each row represents a path of the Depth First Search on the BDD.

$$\begin{aligned} &\mathrm{F}_v + \\ &(1 - \mathrm{F}_V) * \mathrm{F}_{M_1} * \mathrm{F}_{M_2} + \\ &(1 - \mathrm{F}_V) * \mathrm{F}_{M_1} * (1 - \mathrm{F}_{M_2}) * \mathrm{F}_{M_3} + \\ &(1 - \mathrm{F}_V) * (1 - \mathrm{F}_{M_1}) * \mathrm{F}_{M_2} * \mathrm{F}_{M_3} \end{aligned} \tag{4}$$

The technique described above can also be used to carry out symbolic evaluation, i.e., compute the reliability function in analytical form. In particular, each parameter of this function is a symbolic variable representing the failure probability of a single component.

This technique allows the designer to cope with complex architectural patterns without loss of precision, especially

considering that the manual analysis of such structures can rapidly become impractical. As an example, Equation 5 represents the reliability function computed for the architecture with module V111 (Figure 2a), followed by V100 (Figure 2d)[1].

$$\begin{aligned}\mathrm{F}_{sys}(\mathrm{F}_M,\mathrm{F}_V) =& \mathrm{F}_V + (2*\mathrm{F}_M*\mathrm{F}_V) + (6*\mathrm{F}_M^2) - (16*\mathrm{F}_M^4*\mathrm{F}_V^2) - (4*\mathrm{F}_M^3) + \\ &- (10*\mathrm{F}_V*\mathrm{F}_M^2) - (4*\mathrm{F}_M^6*\mathrm{F}_V^2) - (2*\mathrm{F}_M*\mathrm{F}_V^2) - (4*\mathrm{F}_M^6) + \ldots \\ &+ (25*\mathrm{F}_V*\mathrm{F}_M^4) + (12*\mathrm{F}_M^5) - (26*\mathrm{F}_V*\mathrm{F}_M^5) + (4*\mathrm{F}_M^2*\mathrm{F}_V^2) \end{aligned} \tag{5}$$

This formula has been obtained automatically by using symbolic computation techniques based on Equation 3. Computing the symbolic reliability function allows us to compare different architectural configurations independently of the specific values of failure probability. Moreover, the generation of the parametric reliability function allows us to evaluate different modules that implement the same behavior. As an example, let us consider three different modules, M_1, M_2 and M_3, which provide the same capability in terms of functional computation but using different implementations. In this scenario, the symbolic computation allows us to express dependencies between failure probability of different modules. For instance, a setting where the probability of failure of M_1 is F_{M_1}, $\mathrm{F}_{M_2} = 7/8 * \mathrm{F}_{M2}$, and $\mathrm{F}_{M_3} = 5/8 * \mathrm{F}_{M1}$, can be easily expressed in order to evaluate the overall reliability. Equation 6 shows an example of the generated reliability formula, where the failure probability of M_1 is k times the failure of other modules (denoted F_M).

$$\begin{aligned}\mathrm{F}_{sysK}(\mathrm{F}_M,\mathrm{F}_V,k) =& \mathrm{F}_V + (2*\mathrm{F}_M*\mathrm{F}_V) + (2*\mathrm{F}_M^2) - (4*\mathrm{F}_V*\mathrm{F}_M^2) + (4*k*\mathrm{F}_M^5) + \\ &- (4*\mathrm{F}_M^4*k^2) - (4*\mathrm{F}_M^6*k^2) - (2*\mathrm{F}_M*\mathrm{F}_V^2) - (2*\mathrm{F}_M^4*\mathrm{F}_V^2) + \ldots \\ &+ (3*\mathrm{F}_V*\mathrm{F}_M^4) + (14*\mathrm{F}_V*k*\mathrm{F}_M^4) - (10*\mathrm{F}_V*k*\mathrm{F}_M^5) \end{aligned} \tag{6}$$

Most importantly, the reliability function as in Equation 5 takes as input a symbol that should express the failure probability of a specific component (type), i.e., a value in $\mathbb{R}^{[0,1]}$. In principle, assuming the independence between basic events, the reliability function can be extended in order to consider other parameters such as the time interval, giving the probability of failure of the basic events as probability distributions e.g., Weibull or exponential. This approach is described in Equation 7 where $\mathrm{F}_{sysT} : ((\mathbb{R} \mapsto \mathbb{R}^{[0,1]}) \times (\mathbb{R} \mapsto \mathbb{R}^{[0,1]}) \times \mathbb{R}) \mapsto \mathbb{R}$, and both F_M and F_V are functions that respectively compute the probability of failure of M and V, given an interval of time Δ_T.

$$\mathrm{F}_{sysT}(\mathrm{F}_M,\mathrm{F}_V,\Delta_T) \doteq \mathrm{F}_{sys}(\mathrm{F}_M(\Delta_T),\mathrm{F}_V(\Delta_T)) \tag{7}$$

6.4.1. Symbolic and numerical functions

While the concept behind the reliability function computation is described in the Equation 3, its actual implementation exploits the DAG-like structure of BDDs to minimize the generated formula. Algorithm 1 shows this process, which requires as input a BDD, a map from BDD-nodes to symbols representing the variables that identify the probability of failure, and an hashtable used to cache the already computed results. The output of this algorithm is the actual closed form of the reliability function. Internally, Algorithm 1 relies on the generation of a "shared formula" (i.e., line 10), which guarantees to share the common sub parts.

The next step, after the application of the Algorithm 1, is to print out the structure of the shared formula. Figure 11 shows an extract of such output for the architecture composed of the TMR V111 (Figure 2a), followed by the V100 (Figure 2d). In this case, the function "fault_probability" takes as input the probability of failure of each voter (e.g., `TV111V1f` for Voter 1 in TMR V111), and each module (e.g., `TV100M2f` for Module 2 in TMR V111) and provides as output the probability of failure `prob` of the entire architecture.

[1] For brevity, in Equations 5 and 6 we omit some summands, and replace them with ... instead. Moreover, we assume the three modules have the same failure probability, denoted F_M.

Algorithm 1: Symbolic probability computation.

Input: BDD (n), Symbols map ($\mathcal{S}$), Hashtable ($cache = \{\}$)
Result: Probability
if n **in** $cache$ **then**
  return $cache[n]$;
if n = $\top$ **then**
  return *1.0*;
if n = $\bot$ **then**
  return *0.0*;
pthen $\leftarrow$ Probability_computation(get_then_node(n), $\mathcal{S}$, $cache$);
pelse $\leftarrow$ Probability_computation(get_else_node(n), $\mathcal{S}$, $cache$);
pcur $\leftarrow$ $\mathcal{S}$(get_var(n));
$cache[n] \leftarrow shared\_formula$(pcur $\cdot$ pthen + (1.0 $-$ pcur) $\cdot$ pelse);
return $cache[n]$;

```
function prob = fault_probability(TV111V1f, TV111M1f, TV111M2f, TV111M3f, TV100V1f,
    TV100M1f, TV100M2f, TV100M3f)
  EXPR1 = TV111V1f;
  EXPR2 = 1;
  EXPR3 = EXPR2 - EXPR1;
  EXPR4 = TV100V1f;
  EXPR5 = TV100M3f;
  EXPR6 = EXPR2 - EXPR5;
  EXPR7 = TV100M2f;
  EXPR8 = EXPR2 - EXPR7;
  EXPR9 = TV111M3f;
  EXPR10 = TV111M2f;
  ...
  prob = EXPR_40;
```

Fig. 11. Reliability function of the architecture V111 (Figure 2a), followed by V100 (Figure 2d)

7. Computing Cut Sets via predicate abstraction

The main computational bottleneck of the flow described in previous Section is the construction of the set of Minimal Cut Sets. The problem can be seen as an AllSMT(EUF) problem, i.e. the enumeration of all minimal solutions of a formula using AllSMT [LNO06]. Since the performance of available solvers is directly related to the number of minimal cut sets, in realistic cases this computation can be very expensive. In this Section, we present a method to overcome this bottleneck. We first informally present the approach. Then, we define a formal representation for architectures (Sec. 7.2), prove a general equivalence theorem between concrete and abstract architectures (Sec. 7.3), and then specialize it to the case of abstract miter (Sec. 7.4).

7.1. Overview of the approach

We use predicate abstraction to reformulate the problem into an equivalent representation, where the scope of theory variables is localized. This allows us to "push-in" the quantifiers, and partition a global AllSMT(EUF) computation into a number of smaller AllSMT(EUF) problems (in the computation of the abstract modules). The outcome of this process is a Boolean formula that can be quantified with an efficient, BDD-based procedure.

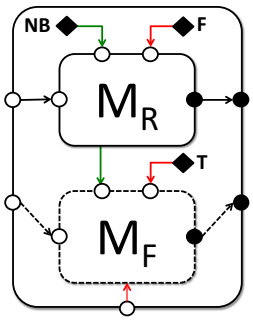


Fig. 12: Stage composition

Our approach is generic, in that it works for a generic set of predicates, not only when the predicates represent component faults. Moreover, since we generate a Boolean formula which is equivalent to the original one, the compositional approach does not impact the overall flow of the analysis, in particular the generation of the reliability function is identical as in the monolithic case, and the Minimal Cut Sets computation is based on the techniques described in [BCGM15]. Also, our approach does not require a specific algorithm to compute the unreliability and/or the fault tree – alternative methods can be used to solve the obtained Boolean formula, e.g. it could be possible to skip the generation of the fault tree and directly compute the unreliability function.

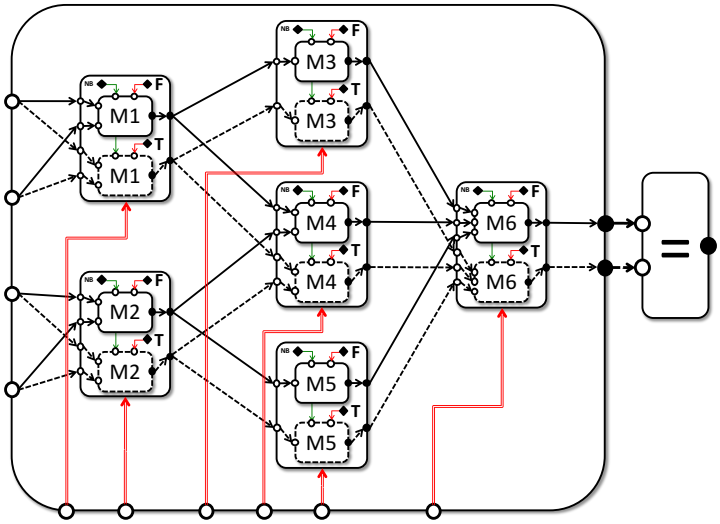


Fig. 13. Stage-based Miter

Stage-based miter

Consider the miter construction presented in Figure 9. It relies on the "black box" comparison of two identical copies of the architecture: the one under analysis, which can fail, and the reference architecture that is constrained not to fail. We notice that every (fallible) module in the architecture under analysis has a corresponding infallible module in the reference architecture. Thus, we consider a miter construction based on a tighter aggregation of the corresponding (faulty and reference) copies of the same module. This aggregation, called a *stage*, is depicted in Figure 12. This construction is generalized at the system level to obtain a stage-based miter. Figure 13 depicts the stage-based miter for the system in Figure 3. Although the two miter constructions are logically equivalent, the stage-based miter combines reference and faulty components in the same block. This characteristic allows us to emphasize the localization of the deviation of a component from its nominal behavior.

Optimized Cut Set computation

The second step is to apply a predicate abstraction on the input and output ports of each stage. Intuitively, the predicates characterize the ways in which the outputs of the component can deviate from the nominal case, given the internal faults and the deviations in the inputs deriving from upstream faults. The abstraction is explicitly represented by means of additional components connected to the input and output ports of each stage, as represented in Figure 14. The C_i components, called *concretizers*, receive as input an assignment to the predicates, and provide as output an instance of concrete signals satisfying them. Analogously, the A_i components, called *abstractors*, give as output the assignment to the predicates corresponding to the concrete data in input. We obtain an architecture that has the same interface as the *concrete* one by adding an abstractor that preprocesses the inputs. This model, depicted in Figure 14, is called *abstract miter*. The fundamental property of the *abstract* miter is that, under some preconditions, it has the very same cut sets as the *concrete* one. We obtain a pure Boolean model by replacing each abstract stage (as in Figure 14) with a Boolean component over the input and output predicate variables, computed by means of a local AllSMT(EUF) call.

7.2. Formal description of architectures

7.2.1. Architectures as combinatorial components

We now introduce a formal notation to describe architectures. The formalism allows us to model combinatorial components, i.e. that do not have persistent state. A *Basic* Combinatorial Component models a system with input and output ports, a set of faults signals, and an SMT(EUF) formula.

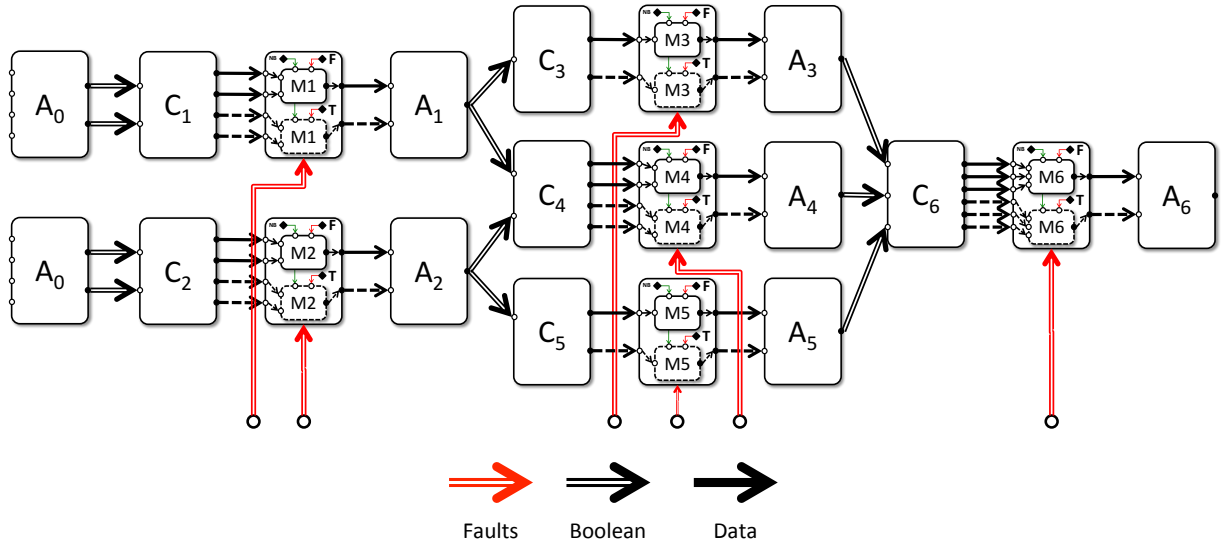


Fig. 14. The abstract Miter

Definition 7.1 (Basic combinatorial component). A basic combinatorial component is a tuple $\langle \vec{I}, \vec{O}, \vec{F}, \pi \rangle$, where:

- $\vec{I}$ is the vector of input ports
- $\vec{O}$ is the vector of output ports
- $\vec{F}$ is the set of faults events
- $\pi(\vec{I}, \vec{O}, \vec{F})$ is an SMT formula.

We denote with $\vec{I}[i]$ the i-th input port, and with $\vec{O}[i]$ the i-th output port. Each port p is associated with a type $\tau(p)$, which can either be Boolean ($\mathbb{B}$) or Data ($\mathbb{D}$). We denote with $\tau(\vec{I})$ the vector of types for the input ports; similarly for $\tau(\vec{O})$. We require $\tau(f) = \mathbb{B}$ for all fault event $f \in \vec{F}$.

More complex combinatorial components are recursively obtained from basic components by means of two composition operators: *sequential composition* ($\triangleright$), and *parallel composition* ($|$).

Sequential composition links the outputs of the first component to the inputs of the second one. It requires that the components are sequentially compatible, i.e. the connected port vectors have the same type, as expressed in Definition 7.2.

Definition 7.2 (Sequential compatibility). Let $M_1 = \langle \vec{I}_1, \vec{O}_1, \vec{F}_1, \pi_1 \rangle$ and $M_2 = \langle \vec{I}_2, \vec{O}_2, \vec{F}_2, \pi_2 \rangle$ be two combinatorial components. M_1 and M_2 are sequentially compatible, denoted $M_1 \rightsquigarrow M_2$, iff $\tau(\vec{O}_1) = \tau(\vec{I}_2)$.

Definition 7.3 formalizes the sequential composition of two components M_1 and M_2. The idea is to connect the output ports of M_1 to the input ports of M_2. The resulting component M has the same input ports as M_1, the same output ports of M_2 and the union of the faults of M_1 and M_2.

Definition 7.3 (Sequential composition). Let $M_1 = \langle \vec{I}_1, \vec{O}_1, \vec{F}_1, \pi_1 \rangle$ and $M_2 = \langle \vec{I}_2, \vec{O}_2, \vec{F}_2, \pi_2 \rangle$ be two combinatorial components. If $M_1 \rightsquigarrow M_2$, their sequential composition $M \doteq M_1 \triangleright M_2$ is defined as $\langle \vec{I}_1, \vec{O}_2, \vec{F}_1 \cup \vec{F}_2, \pi \rangle$ where

$$\pi \doteq \exists \vec{O}_1, \vec{I}_2 : \pi_1(\vec{I}_1, \vec{O}_1, \vec{F}_1) \wedge \pi_2(\vec{I}_2, \vec{O}_2, \vec{F}_2) \wedge \vec{O}_1 = \vec{I}_2$$

Parallel composition, on the other hand, juxtaposes the set of ports of the two components, which run in parallel. We require the set of faults of the components to be disjoint.

Definition 7.4 (Parallel composition). Let $M_1 = \langle \vec{I}_1, \vec{O}_1, \vec{F}_1, \pi_1 \rangle$ and $M_2 = \langle \vec{I}_2, \vec{O}_2, \vec{F}_2, \pi_2 \rangle$ be two combinatorial

components such that $\vec{F}_1 \cap \vec{F}_2 = \emptyset$. Their parallel composition $M_1|M_2$ is defined as

$$\langle\ \vec{I}_1 \cdot \vec{I}_2,\ \vec{O}_1 \cdot \vec{O}_2,\ \vec{F}_1 \cup \vec{F}_2,\ \pi_1(\vec{I}_1, \vec{O}_1, \vec{F}_1) \wedge \pi_2(\vec{I}_2, \vec{O}_2, \vec{F}_2)\ \rangle$$

where $\cdot$ represents vector juxtaposition.

This framework enables the definition of any tree- or DAG-shaped structure. In the following, we assume that the components are expressed in form of sequential composition of *layers*, where each layer is the parallel composition of basic components. This is without loss of generality: specialized basic components can be used to connect inputs and outputs ports in a different fashion than just parallel and sequential to adjacent components. To this end, we identified three classes of components: duplication of values (D), propagation of input values (I, for identity), and arbitrary reconfiguration of signals (R module).

Equation 8 represents the system in Figure 3 as a combinatorial component. In this case, we use a D component in order to duplicate outputs of the M_1 and M_2 components.

$$(M_1|M_2) \triangleright (D|D) \triangleright (M_3|M_4|M_5) \triangleright M_6 \tag{8}$$

Example 7.1 provides a possible representation of the R module.

Example 7.1 (Reconfiguration as a basic component). A reconfiguration module R is a basic component with two Real inputs i_1, i_2 and two Real outputs o_1, o_2, where i_1 is linked to o_2 and i_2 to o_1. Formally, $R \doteq \langle \vec{I}, \vec{O}, \vec{F}, \pi \rangle$ where:

- $\vec{I} = [i_1, i_2]$;
- $\vec{O} = [o_1, o_2]$;
- $\vec{F} = \emptyset$;
- $\pi(\vec{I}, \vec{O}, \vec{F}) = (i_1 = o_2) \wedge (i_2 = o_1)$.

$\diamond$

Example 7.2 (TMR V111 as a combinatorial component). Giving a computational module with one input and one output, whose nominal and faulty behaviors are respectively defined by the uninterpreted functions $NB_M : \mathbb{D} \to \mathbb{D}$, and $FB_M : \mathbb{D} \to \mathbb{D}$, and a faulty voter behavior $FB_V : \mathbb{D} \times \mathbb{D} \times \mathbb{D} \to \mathbb{D}$, then the corresponding Triple Modular Redundancy with one voter (V111 as in Figure 6) is represented as a combinatorial component $\langle \vec{I}, \vec{O}, \vec{F}, \pi \rangle$ such that:

- $\vec{I} = [i_1, i_2, i_3]$;
- $\vec{O} = [o_1, o_2, o_3]$;
- $\vec{F} = \{f_{M1}, f_{M2}, f_{M3}, f_V\}$;
- $\pi(\vec{I}, \vec{O}, \vec{F}) = \exists NB_M, FB_M, FB_V : (\bigwedge_{i=1..3} o_i = \textsc{Tmr}(\vec{I}, \vec{F}, NB_M, FB_M, FB_V))$.

and where the formula TMR is defined as:

$$\begin{aligned}
\textsc{Tmr}(\vec{I}, \vec{F}, NB_M, FB_M, FB_M, FB_V) &\doteq \textsc{Voter}(M(i_1, NB_M, FB_M, f_{M1}),\\
&\qquad M(i_2, NB_M, FB_M, f_{M2}),\\
&\qquad M(i_3, NB_M, FB_M, f_{M3}), FB_V, f_V)\\
M(i, NB_M, f_M) &\doteq \textsc{if}\ \neg f_M\ \textsc{then}\ NB_M(i)\ \textsc{else}\ FB_M(i)\\
\textsc{Voter}(i_1, i_2, i_3, FB_V, f_V) &\doteq \textsc{if}\ \neg f_V\ \textsc{then}\ NB_V(i_1, i_2, i_3)\ \textsc{else}\ FB_V(i_1, i_2, i_3)\\
NB_V(i_1, i_2, i_3) &\doteq \textsc{if}\ i_1 = i_2\ \textsc{then}\ i_1\ \textsc{else}\ e_1\\
e_1 &\doteq \textsc{if}\ i_1 = i_3\ \textsc{then}\ i_1\ \textsc{else}\ e_2\\
e_2 &\doteq \textsc{if}\ i_2 = i_3\ \textsc{then}\ i_2\ \textsc{else}\ i_3
\end{aligned}$$

Intuitively, each fault variable f_{Mi}, when set to false, binds the output of each module to the output of the function NB_M, while they leave them free when assigned to true. The voters are described in a similar way, but in this case the behavior is explicitly defined as NB_V. $\diamond$

The structures defined via the combinatorial components can be proven to be equivalent if they satisfy the Definition 7.5. According to that, we can now define a set of important properties that characterize these structures. In

particular, Lemma 7.1 states that if two combinatorial components are equivalent, it is possible to sequentially combine them with a third component and preserve the equivalence. Lemma 7.2 states a similar result for parallel composition. Lemma 7.3 shows that the application of sequential and parallel composition can be inverted when the sequential compatibility allows for the former composition to be applied. Another important property is the associativity (i.e., Lemma 7.4), which applies only for the parallel composition, while commutativity does not apply for either of the operators.

Definition 7.5 (Equivalence). Two combinatorial components $M_1 = \langle \vec{I}_1, \vec{O}_1, \vec{F}_1, \pi_1 \rangle$ and $M_2 = \langle \vec{I}_2, \vec{O}_2, \vec{F}_2, \pi_2 \rangle$, such that $\vec{F}_1 = \vec{F}_2$, $\vec{I}_1 = \vec{I}_2$, and $\vec{O}_1 = \vec{O}_2$, are called equivalent, denoted $M_1 \equiv M_2$, if and only if π_1 and π_2 are logically equivalent.

Lemma 7.1 (Sequential equivalence). Given the combinatorial components M, M_1, and M_2, if $M_1 \equiv M_2$, $M \rightsquigarrow M_1 \rightsquigarrow M$, and $M \rightsquigarrow M_2 \rightsquigarrow M$, then $M \triangleright M_1 \equiv M \triangleright M_2$ and $M_1 \triangleright M \equiv M_2 \triangleright M$.

Lemma 7.2 (Parallel equivalence). Given the combinatorial components M_1, M_2, M_3, M_4, if $M_1 \equiv M_2$ and $M_3 \equiv M_4$ then $M_1|M_3 \equiv M_2|M_4$.

Lemma 7.3 (Inversion equivalence). Given the combinatorial components M_1, M_2, M_3, M_4, if $M_1 \rightsquigarrow M_2$ and $M_3 \rightsquigarrow M_4$ then $(M_1|M_3) \triangleright (M_2|M_4) \equiv (M_1 \triangleright M_2)|(M_3 \triangleright M_4)$.

Lemma 7.4 (Associativity). Given the combinatorial components M_1, M_2, and M_3, if $M_1 \rightsquigarrow M_2$ and $M_2 \rightsquigarrow M_3$ then $(M_1 \triangleright M_2) \triangleright M_3 \equiv M_1 \triangleright (M_2 \triangleright M_3)$ and $(M_1|M_2)|M_3 \equiv M_1|(M_2|M_3)$.

7.3. Equivalence modulo abstraction

We now define two special types of combinatorial components, whose purpose is to support the modeling of abstraction. *Abstractor components* (Definition 7.6) are used to translate a set of concrete (data) values into their abstract counterpart, whereas *Concretizer components* (Definition 7.7) generate instances of concrete values satisfying the predicates.

Definition 7.6 (Abstractor). A combinatorial component $A = \langle \vec{I}, \vec{O}, \vec{F}, \alpha \rangle$ is an abstractor iff $\tau(o) = \mathbb{B}$ for all $o \in \vec{O}$, and $\vec{F} = \emptyset$.

Definition 7.7 (Concretizer). A combinatorial component $C = \langle \vec{I}, \vec{O}, \vec{F}, \gamma \rangle$ is a concretizer iff $\tau(i) = \mathbb{B}$ for all $i \in \vec{I}$, and $\vec{F} = \emptyset$.

By way of abstractors and concretizers, we can express the abstraction of a component M as the sequential composition $C \triangleright M \triangleright A$.

Example 7.3 (Module abstraction via abstractors and concretizers). Given a combinatorial component M, with two input ports $\vec{I}_M = [M.i, M.i']$, two output ports $\vec{O}_M = [M.o, M.o']$, one fault $\vec{F} = [M.f]$, and a behavior $\pi \doteq (M.i = M.o) \wedge (\neg M.f \rightarrow (M.i' = M.o'))$. The ports whose name is primed (e.g., $M.i'$) are called faulty, while the not primed ports are the reference ones. According to that, the behavior π defines a connection between reference ports (i.e., $M.i$ and $M.o$), while the faulty ones (i.e., $M.i'$ and $M.o'$) are forced to be equal only when $M.f$ is false.

Assuming that we abstract the behavior π with the predicates Pi and Po, such that $Pi \iff (M.i = M.i')$ and $Po \iff (M.o = M.o')$, then the resulting abstracted behavior π^A is defined as:

$$\pi^A \doteq \exists M.i, M.o, M.i', M.o' : \pi \wedge (Pi \iff (M.i = M.i')) \wedge (Po \iff (M.o = M.o'))$$

which simplifies to:

$$\pi^A \doteq Pi \rightarrow (\neg M.f \rightarrow Po)$$

The application of the technique described in this paper requires the definition of concretizer and abstractor components. In this particular case, they are respectively $C = \langle \vec{I}_C, \vec{O}_C, \emptyset, \gamma \rangle$ and $A = \langle \vec{I}_A, \vec{O}_A, \emptyset, \alpha \rangle$, where $\vec{I}_C = [Pi]$, $\vec{O}_C = [C.o, C.o']$, $\vec{I}_A = [A.i, A.i']$, and $\vec{O}_A = [Po]$. The definition of γ and α behaviors follow the predicates expressing the equality between reference and faulty ports, thus $\gamma \doteq Pi \iff (C.o = C.o')$ and $\alpha \doteq Po \iff (A.i = A.i')$.

Since the condition $C \rightsquigarrow M \rightsquigarrow A$ holds, the abstraction of component M can be defined as $C \triangleright M \triangleright A$. The

behavior π^{CMA} of the combinatorial component $C \triangleright M \triangleright A = \langle \vec{I}_C, \vec{O}_A, \vec{F}, \pi^{CMA} \rangle$, resulting from the application of the sequential composition, is defined as follows:

$$\begin{aligned}
\pi^{CMA} \doteq & (\exists \vec{O}_C, \vec{I}_M, \vec{O}_M, \vec{I}_A : \gamma \wedge \pi \wedge \alpha \wedge \\
& (C.o = M.i) \wedge (C.o' = M.i') \wedge (M.o = A.i) \wedge (M.o' = A.i')) \\
\equiv & (\exists \vec{O}_C, \vec{I}_M, \vec{O}_M, \vec{I}_A : \\
& (Pi \iff (C.o = C.o')) \wedge \pi \wedge (Po \iff (A.i = A.i')) \wedge \\
& (C.o = M.i) \wedge (C.o' = M.i') \wedge (M.o = A.i) \wedge (M.o' = A.i')) \\
\equiv & (\exists \vec{I}_M, \vec{O}_M : \\
& (Pi \iff (M.i = M.i')) \wedge \pi \wedge (Po \iff (M.o = M.o'))) \\
\equiv & (Pi \rightarrow (\neg M.f \rightarrow Po))
\end{aligned}$$

The result expresses that π^{CMA} is equivalent to π^A.

◇

We now prove that, under suitable conditions, a system composed of concrete modules is equivalent to a system where each individual module is replaced with its abstract counterpart.

Theorem 7.1 (Modular abstraction equivalence). For all $i \in \{1..n\}$, for all $j \in \{1..m_i\}$ let $M_{i,j}$ be combinatorial components and let $C_{i,j}$ be concretizers. For all $i \in \{1..(n+1)\}$, for all $j \in \{1..m_i\}$ let $A_{i,j}$ be abstractors. Let $C_{i,j} \rightsquigarrow M_{i,j}$ and $M_{i,j} \rightsquigarrow A_{i,j}$. Let

$$\begin{array}{lcccccccccccc}
\text{CONCRETESYS} & \doteq & & & L_n & \triangleright & \ldots & \triangleright & L_2 & \triangleright & L_1 & \triangleright & A_1 \\
\text{ABSTRACTSYS} & \doteq & A_{n+1} & \triangleright & CLA_n & \triangleright & \ldots & \triangleright & CLA_2 & \triangleright & CLA_1 & &
\end{array}$$

where

$$L_i = \left(\begin{array}{c} M_{i,1} \\ \hline M_{i,2} \\ \hline \cdots \\ \hline M_{i,m_i} \end{array} \right), C_i = \left(\begin{array}{c} C_{i,1} \\ \hline C_{i,2} \\ \hline \cdots \\ \hline C_{i,m_i} \end{array} \right), A_i = \left(\begin{array}{c} A_{i,1} \\ \hline A_{i,2} \\ \hline \cdots \\ \hline A_{i,m_i} \end{array} \right), CLA_i = C_i \triangleright L_i \triangleright A_i$$

If for all $i \in \{1..n\}$, for all $j \in \{1..m_i\}$ it holds that

$$A_{i+1,j} \triangleright C_{i,j} \triangleright M_{i,j} \triangleright A_{i,j} \quad \equiv \quad M_{i,j} \triangleright A_{i,j} \tag{9}$$

then

$$\text{CONCRETESYS} \equiv \text{ABSTRACTSYS}$$

Theorem 7.1 allows us to generate an equivalent network of combinatorial components by using only Boolean modules. Namely, it enables substitution of a concrete module with its abstract counterpart, provided that the application of abstraction and concretization on inputs preserves the behavior of the outputs in the abstract domain, as formally defined by the hypothesis. CONCRETESYS and ABSTRACTSYS systems in Theorem 7.1 are sequential compositions of layers, and each layer is a parallel composition of combinatorial components.

Proof. The proof proceeds by induction on the number of layers (sequential composition), and on the number of elements within a layer (parallel composition).

Parallel composition

First, we need to prove that if Equation 9 holds then that property can be lifted to layers. For all $i \in \{1, \ldots, n\}$

$$L_i \triangleright A_i \equiv A_{i+1} \triangleright C_i \triangleright L_i \triangleright A_i \tag{10}$$

Parallel composition: base case

When $m_i = 1$, we have $M_1 \triangleright A_1 \equiv A_2 \triangleright C_1 \triangleright M_1 \triangleright A_1$, which follows from the hypothesis of the theorem.

Parallel composition: step case

Assuming that Equation 10 holds for n, then if $M_{i,n+1} \triangleright A_{i,n+1} \equiv A_{i+1,n+1} \triangleright C_{i,n+1} \triangleright M_{i,n+1} \triangleright A_{i,n+1}$ by Lemma 7.2 we obtain

$$\frac{\left(\begin{array}{c}M_{i,1}\\ \hline M_{i,2}\\ \hline \cdots \\ \hline M_{i,n}\end{array}\right) \triangleright \left(\begin{array}{c}A_{i,1}\\ \hline A_{i,2}\\ \hline \cdots \\ \hline A_{i,n}\end{array}\right)}{M_{i,n+1} \triangleright A_{i,n+1}} \equiv \frac{\left(\begin{array}{c}A_{i+1,1}\\ \hline A_{i+1,2}\\ \hline \cdots \\ \hline A_{i+1,n}\end{array}\right) \triangleright \left(\begin{array}{c}C_{i,1}\\ \hline C_{i,2}\\ \hline \cdots \\ \hline C_{i,n}\end{array}\right) \triangleright \left(\begin{array}{c}M_{i,1}\\ \hline M_{i,2}\\ \hline \cdots \\ \hline M_{i,n}\end{array}\right) \triangleright \left(\begin{array}{c}A_{i,1}\\ \hline A_{i,2}\\ \hline \cdots \\ \hline A_{i,n}\end{array}\right)}{A_{i+1,n+1} \triangleright C_{i,n+1} \triangleright M_{i,n+1} \triangleright A_{i,n+1}} \tag{11}$$

then, considering that the hypothesis of the Theorem guarantees a sequential compatibility of the sequential compositions, by Lemma 7.3 and Lemma 7.4 it is possible to switch the top sequential and parallel composition on both sides of the equivalence, and conclude that Equation 12 holds.

$$\left(\begin{array}{c}M_{i,1}\\ \hline \cdots \\ \hline M_{i,n}\\ \hline M_{i,n+1}\end{array}\right) \triangleright \left(\begin{array}{c}A_{i,1}\\ \hline \cdots \\ \hline A_{i,n}\\ \hline A_{i,n+1}\end{array}\right) \equiv \left(\begin{array}{c}A_{i+1,1}\\ \hline \cdots \\ \hline A_{i+1,n}\\ \hline A_{i+1,n+1}\end{array}\right) \triangleright \left(\begin{array}{c}C_{i,1}\\ \hline \cdots \\ \hline C_{i,n}\\ \hline C_{i,n+1}\end{array}\right) \triangleright \left(\begin{array}{c}M_{i,1}\\ \hline \cdots \\ \hline M_{i,n}\\ \hline M_{i,n+1}\end{array}\right) \triangleright \left(\begin{array}{c}A_{i,1}\\ \hline \cdots \\ \hline A_{i,n}\\ \hline A_{i,n+1}\end{array}\right) \tag{12}$$

Sequential composition

We now prove, for all n, that

$$\begin{array}{ccccccccc} & & L_n & \triangleright & L_{n-1} & \triangleright & \ldots & \triangleright & L_1 \triangleright A_1 \\ A_{n+1} & \triangleright & (C_n \triangleright L_n \triangleright A_n) & \triangleright & (C_{n-1} \triangleright L_{n-1} \triangleright A_{n-1}) & \triangleright & \ldots & \triangleright & (C_1 \triangleright L_1 \triangleright A_1) \end{array} \equiv \tag{13}$$

Sequential composition: base case

When $n = 1$, $L_1 \triangleright A_1 \equiv A_2 \triangleright C_1 \triangleright L_1 \triangleright A_1$ follows directly from Equation 10.

Sequential composition: step case

Assume that the property of Equation 13 holds for n. By Lemma 7.1 it is possible to prepend L_{n+1}:

$$\begin{array}{c} L_{n+1} \triangleright L_n \triangleright \ldots \triangleright L_1 \triangleright A_1 \equiv \\ L_{n+1} \triangleright A_{n+1} \triangleright C_n \triangleright L_n \triangleright A_n \triangleright C_{n-1} \triangleright \ldots \triangleright L_1 \triangleright A_1 \end{array} \tag{14}$$

Then, if $L_{n+1} \triangleright A_{n+1} \equiv A_{n+2} \triangleright C_{n+1} \triangleright L_{n+1} \triangleright A_{n+1}$ by Lemma 7.1 we obtain

$$\begin{array}{c} L_{n+1} \triangleright A_{n+1} \triangleright C_n \triangleright L_n \triangleright A_n \triangleright C_{n-1} \triangleright \ldots \triangleright L_1 \triangleright A_1 \equiv \\ A_{n+2} \triangleright C_{n+1} \triangleright L_{n+1} \triangleright A_{n+1} \triangleright C_n \triangleright L_n \triangleright A_n \triangleright C_{n-1} \triangleright \ldots \triangleright L_1 \triangleright A_1 \end{array} \tag{15}$$

and using Equation 14 we obtain

$$\begin{array}{ccccccc} & L_{n+1} & \triangleright & L_n & \triangleright & \ldots \triangleright & L_1 \triangleright A_1 \\ A_{n+2} \triangleright & C_{n+1} \triangleright L_{n+1} \triangleright A_{n+1} & \triangleright & C_n \triangleright L_n \triangleright A_n & \triangleright & \ldots \triangleright & C_1 \triangleright L_1 \triangleright A_1 \end{array} \equiv \tag{16}$$

which proves that the property of Equation 13 holds also for $n + 1$. $\square$

7.4. Specialization to miter equivalence

We now discuss how to apply Theorem 7.1 to the problem of computing the set of cut sets for TLE in the concrete miter, and reduce it to the problem of computing the cut sets of an equivalent, abstract Boolean system.

7.4.1. Model definition

In the context of the architecture analysis, each module $M_{i,j}$ of CONCRETESYS and ABSTRACTSYS is a stage composition as in Def. 7.8 and represented in Fig. 12.

Definition 7.8 (Stage). Let M be $\langle \vec{I}, \vec{O}, \vec{F}, \pi \rangle$. Its stage composition $S = \langle \vec{I}_S, \vec{O}_S, \vec{F}_S, \pi_S \rangle$ is the parallel composition $M|M'$, where $M' = \langle \vec{I}', \vec{O}', \emptyset, \pi' \rangle$ is the reference for M, defined as

- $\vec{I}' \doteq \bigcup_{p \in \vec{I}} p'$;
- $\vec{O}' \doteq \bigcup_{p \in \vec{O}} p'$;
- $\pi' \doteq \exists \vec{F} : \pi[\vec{I}/\vec{I}'; \vec{O}/\vec{O}'] \wedge \bigwedge_{f \in \vec{F}} \neg f$.

As shown in Figure 13, the miter construction consists in linking input ports of reference ($\vec{I}_R$) and faulty ($\vec{I}_F$) architectures, and provide a TLE signal, expressed over the Boolean outputs $\vec{O}$ (i.e., the equality between the outputs of reference and fault architectures). As a result, the formula representing the miter construction is as follows:

$$\text{TLE} \doteq \bigvee_{o \in \vec{O}} \neg o, \ \text{EQ} \doteq \bigwedge_{i \in \{1..|\vec{I}_R|\}} \vec{I}_R[i] = \vec{I}_F[i]$$

$$\text{MITER} \doteq \text{TLE} \wedge \text{EQ}$$

Given that CONCRETESYS is a formula over $\vec{I} = \vec{I}_R | \vec{I}_F$ and $\vec{O}$, the resulting cut set computation for TLE is

$$\exists \vec{I}, \vec{O} : (\text{CONCRETESYS} \wedge \text{MITER})$$

The result is a Boolean formula in the $\vec{F}_1 \ldots \vec{F}_n$ variables. The idea is to replace CONCRETESYS with an equivalent ABSTRACTSYS, and to exploit its structure to optimize the computation. Specifically, we consider a specialized class of components, including the typical TMR constructs (described in Figure 2), and a class of abstractors and concretizers induced by a set of stage-specific predicates.

7.4.2. Stage abstraction and concretization

Predicate abstraction is a form of abstraction where the concrete state space is partitioned according to the assignments induced on a set of predicates $\mathbb{P}$. In this case the concrete space corresponds to the cross product of domains of the port variables and fault events. Given the stage S for the module M, we define the corresponding set of predicates $\mathbb{P} \doteq \mathbb{P}_I \cup \mathbb{P}_O$, where $\mathbb{P}_I \doteq \{(p = p') | p \in \vec{I}\}$, and similarly for $\mathbb{P}_O$. These predicates express the equivalence between the ports of M and the corresponding ports of the *reference* component M'.

The abstractor A_S is defined as

$$\langle \vec{I} \cup \vec{I}', \mathbb{O}, \emptyset, \bigwedge_{p \in \vec{I}} op_{p=p'} \leftrightarrow (p = p') \rangle$$

where $\mathbb{O}$ is the set of Boolean variables $op_{p=p'}$, referred to as (output) predicate names. Similarly, the concretizer C_S, where the predicates $ip_{p=p'} \in \mathbb{I}$, is defined as

$$\langle \mathbb{I}, \vec{O} \cup \vec{O}', \emptyset, \bigwedge_{p \in \vec{O}} ip_{p=p'} \leftrightarrow (p = p') \rangle$$

7.4.3. Optimizing the computation

We follow the notation of Theorem 7.1. For the sake of clarity, we may use the notation of a component to denote its transfer formula. Without loss of generality, we consider the case where layers have only one module, i.e. $m_i = 1$ and

L_i coincides with S_i, i.e. a *stage* composition. In this setting, CLA_i becomes $(C_i \triangleright S_i \triangleright A_i)$. Being CONCRETESYS and ABSTRACTSYS equivalent, we can compute the same result by quantifying out in ABSTRACTSYS the input and output variables of the modules M, the concretizers C and the abstractors A.

$$\exists \vec{I}_n, \vec{I}'_n, \vec{O}_n, \vec{O}'_n, \ldots, \vec{I}_1, \vec{I}'_1, \vec{O}_1, \vec{O}'_1, \mathbb{I}, \mathbb{O} : (A_{n+1} \wedge (C_n \triangleright S_n \triangleright A_n) \wedge \ldots \wedge (C_2 \triangleright S_2 \triangleright A_2) \wedge (C_1 \triangleright S_1 \triangleright A_1) \wedge \text{MITER})$$

Exploiting the structure of ABSTRACTSYS, we can "push in" the quantifiers, thus obtaining

$$\exists \mathbb{I}, \mathbb{O} : (A_{n+1} \wedge \underbrace{(\exists \vec{I}_n, \vec{I}'_n, \vec{O}_n, \vec{O}'_n, : (C_n \triangleright S_n \triangleright A_n))} \wedge \ldots \wedge \underbrace{(\exists \vec{I}_1, \vec{I}'_1, \vec{O}_1, \vec{O}'_1 : (C_1 \triangleright S_1 \triangleright A_1))} \wedge \text{MITER})$$

The underbraces make explicit the quantifications associated with a sequential composition of AllSMT computations. The advantage is to trade one possibly very expensive quantification with many smaller, and possibly factorizable, quantifications. This technique provides a significant advantage when each formula $C_i \triangleright S_i \triangleright A_i$ is a composition of basic components (e.g., $(C_{i1}|C_{i2}) \triangleright (S_{i1}|S_{i2}) \triangleright (A_{i1}|A_{i2})$), since in this case the quantifiers can be pushed in even further. In a sense, the application of the concretization/abstraction around each module has the effect of separating the "theory reasoning". The remaining free variables are the Boolean predicate names, and the remaining quantification can be obtained by standard BDD-based techniques.

7.4.4. Hypothesis discharge and abstraction caching

In order to apply the transformation described above, it is necessary to prove that this class of components satisfies the hypothesis of Theorem 7.1. Specifically, we need to show for all i, j that

$$A_{i+1,j} \triangleright C_{i,j} \triangleright S_{i,j} \triangleright A_{i,j} \qquad \equiv \qquad S_{i,j} \triangleright A_{i,j}$$

for all the stages S obtained from the components M that are used in the architecture under analysis. Interestingly, the check for equivalence can be reduced to the satisfiability of an SMT formula encoding that, under the same inputs, $A_{i+1,j} \triangleright C_{i,j} \triangleright S_{i,j} \triangleright A_{i,j}$ and $S_{i,j} \triangleright A_{i,j}$ can produce different outputs[2].

In order to optimize the overall computation, we also exploit the fact that the transfer functions of the abstract stages remain the same, from problem to problem. Thus the results of the quantification

$$CSA_i \doteq \exists \vec{I}_i \vec{I}'_i \vec{O}_i \vec{O}'_i : (C_i \triangleright S_i \triangleright A_i)$$

for each of the stage compositions of redundant schemas in Figure 2 can be computed once and for all, cached as a library, and reused when needed.

8. Experimental evaluation

8.1. Implementation

The approach described in the previous Sections has been implemented in a tool for reliability analysis of redundant architectures, on top of the following verification backends.

- OCRA [CDT13] is a framework for contract-based design. It supports an architectural specification language, where components are associated with ports of various types, including booleans, reals, uninterpreted objects and functions. In an OCRA architectural decomposition the leaves can be associated with behavioral descriptions expressed in the language of NUXMV. OCRA can generate a monolithic description in the NUXMV language.
- NUXMV is a model checker [CCD+14] for finite- and infinite-state systems. At its core, NUXMV uses the SMT solver MATHSAT [BBC+05, CGSS13], which includes several theories like linear arithmetic over reals and integers, difference logic, bit vectors, uninterpreted functions, and equality. NUXMV provides various SMT-based verification algorithms, and various engines for predicate abstraction [DBL07, DBL09].

[2] Clearly, in order for the equivalence to hold, the formula must be unsatisfiable.

```
COMPONENT NETWORK system
INTERFACE
INPUT PORT in_1 : real;
INPUT PORT in_2 : real;
INPUT PORT in_3 : real;
INPUT PORT in_4 : real;
OUTPUT PORT out : real;
REFINEMENT
sub module_1 : MODULE_2_INPUTS_TYPE;
sub module_2 : MODULE_2_INPUTS_TYPE;
sub module_3 : MODULE_1_INPUT_TYPE;
sub module_4 : MODULE_2_INPUTS_TYPE;
sub module_5 : MODULE_1_INPUT_TYPE;
sub module_6 : MODULE_3_INPUTS_TYPE;
CONNECTION module_1.in_1 := in_1;
CONNECTION module_1.in_2 := in_2;
CONNECTION module_2.in_1 := in_3;
CONNECTION module_2.in_2 := in_4;
CONNECTION module_3.in_1 := module_1.out;
CONNECTION module_4.in_1 := module_1.out;
CONNECTION module_4.in_2 := module_2.out;
CONNECTION module_5.in_1 := module_2.out;
CONNECTION module_6.in_1 := module_3.out;
CONNECTION module_6.in_2 := module_4.out;
CONNECTION module_6.in_3 := module_5.out;
CONNECTION out := module_6.out;

COMPONENT MODULE_1_INPUT_TYPE
INTERFACE
INPUT PORT in_1 : real;
OUTPUT PORT out : real;
PARAMETER nominal_behavior :
                  real -> real;
PARAMETER can_fail : boolean;
COMPONENT MODULE_2_INPUTS_TYPE
INTERFACE
INPUT PORT in_1 : real;
INPUT PORT in_2 : real;
OUTPUT PORT out : real;
PARAMETER nominal_behavior :
                  real -> real;
PARAMETER can_fail : boolean;
COMPONENT MODULE_3_INPUTS_TYPE
INTERFACE
INPUT PORT in_1 : real;
INPUT PORT in_2 : real;
INPUT PORT in_3 : real;
OUTPUT PORT out : real;
PARAMETER nominal_behavior :
                  real -> real;
PARAMETER can_fail : boolean;
```

Fig. 15. The OCRA code for the architecture in Figure 3

- XSAP [BBC+16] is a tool for model based safety analysis built on top of NUXMV. XSAP extends the verification engines of NUXMV to implement various algorithms [BCT07, BCGM15] for safety analysis, including the generation of fault trees and FMEA tables. It supports the synthesis of minimal cut sets, and the extraction of the reliability functions.

Architecture modeling

The language of OCRA is used to specify the architecture under analysis. The user can follow some specific (but non-restrictive) guidelines, leveraging a library of OCRA components, including the TMR in Figure 2 and other basic components. A leaf component of the OCRA architecture is linked to a corresponding SMV implementation. We model each component with a Boolean `can_fail` parameter, which if set to `False` forces the component to behave infallibly, i.e., just like the computing module. The library of abstract components consists of the various different TMR configurations described in Figure 2, the basic identity, voters, and simple computing units of various arity.

The OCRA description for the architecture in Figure 3 is reported in Figure 15. The architecture description requires linking each leaf components to an SMV implementation.

Generation for TLE miter

The miter-based description of the TLE is automatically generated by a dedicated procedure that manipulates the abstract syntax tree of the OCRA model. The description of the architecture under analysis is instantiated twice: for the instance representing the architecture under analysis, the `CanFail` parameters are set to `True`; for the other, representing the reference architecture, `CanFail` parameters all set to `False`. Both replicas are fed with the same uninterpreted function symbols, representing the behavior of the basic modules. The output is also an OCRA file, which is then connected to the suitable implementation modules in NUXMV. Then, the monolithic NUXMV file is generated.

The user can specify which techniques should be used i.e., *concrete*- or *abstraction*-based. In case of the latter, the tool composes the architecture by computing the abstract counterpart of each module. This operation is performed after the correctness check that the hypotheses of Theorem 7.1 are met. Given the size of the modules that we are considering, the hypothesis check requires a negligigle time to be performed. The steps described in Section 7.4.4 rely on the MATHSAT SMT solver [CGSS13]. A memoization mechanism is implemented so that the abstract counterpart of a basic component is computed only once, and then stored into a "cache" folder. During the generation of the abstract architecture, the tool also builds an ordering of the Boolean variables present in the system, in order to optimize the BDD construction. The ordering is used statically, and it is constructed starting from the inputs, and following the architecture topology with an increasing path distance.

Computation of Minimal Cut Sets

The file representing the TLE can be fed directly to XSAP. Two methods are available for the computation of MCSs. The *basic* method is basically a quantifier elimination on the SMT(EUF) formula corresponding to the TLE. Specifically, it is a call to an AllSMT routine [LNO06], where the only remaining variables (i.e., the ones that are not quantified) are the Boolean fault mode variables.

When the *compositional* method is applied, the actual computation corresponds to the construction of the BDD of the set of MCS. This is basically a sequence of projections predicate variables, which is carried out by means of BDD-based projection routines [Som98], proceeding bottom-up. This technique uses the variable ordering information, generated in the previous phase, which seems to give better performance compared to dynamical ordering construction.

Reliability function extraction

The MCS are compacted by exploiting the conversion in form of Binary Decision Diagrams [Bry86]. The construction provides the best performance by disabling dynamic reordering, and using a statically computed ordering, based on the topology of the analyzed system. In detail, considering the example in Expression 8, the ordering starts with faults and output predicates for the module M_1, followed by the variables of M_2, then the ones from M_3 (D modules do not have variables), and so on. The reliability function can be emitted in different formats, including Matlab/Octave and Python. The flow enables detailed analytical evaluations that can be carried out using specific numerical computation software tools. For example, the plots illustrated in Section 6 are obtained automatically from the model of the TMR chains of [HTK10].

8.2. Experimental set up

The setting for the experimental evaluation comprises the generation of the abstract modules, for each of the possible pairs of fallible/reference components in the library represented in Figure 2, and then caching their machine representation. This operation takes on average 5 seconds with a maximum time of 10 seconds. The time needed to initialize the system is not taken into account in the scalability evaluation.

We selected a set of benchmarks of different forms, with increasing size. For each of them, we compare the basic and compositional algorithms for MCS computations. Whenever both techniques terminated, we checked the correctness by comparing the corresponding sets of MCSs.

We ran the experiments on an Intel Xeon E3-1270 at 3.40GHz, with a timeout of 2000 seconds, and a memory limit of 1 GB. All the experiments are available for inspection at `http://es.fbk.eu/people/mattarei/dist/redundant_architectures_analysis`.

We compare two algorithms: the direct application of AllSMT (described in Section 6) and the improvement based on predicate abstraction as described in Section 7. The time for initializing the library in the latter case accounts in total for less than 1 minute. This cost is payed only once, and the necessary abstractions can be cached and are included in the toolset.

8.2.1. Benchmarks

The experiments consider networks of redundant components with different combinations of structures, for a total of 5200 benchmark problems.

In order to evaluate the performance of modular abstraction, we consider the following architectural structures:

- **Linear**: chains of TMR components of the same type, i.e., all TMR V111, V121, or V123;
- **Rectangular**: sequence of a repeating structure composed of 2 TMRs with 1 (triple) input, and 2 TMRs with 2 (triple) inputs. Figure 16 shows a graphical representation of a rectangular structure of length N. The total number of redundant modules for a rectangular system is $4 * N$;
- **Random**: a randomly generated structure with 1 (triple) input and 1 (triple) output. Given a size n, representing the number of redundant components in the system, the maximum diameter of the random structure is bounded to $\sqrt{n}$.

For each family of benchmarks we considered different sizes, thus the (SMT-based) concrete model can range from 19 state variables (i.e., 8 Boolean and 11 Real) to 19206 (i.e., 9600 Boolean and 9606 Real). The detail on each benchmark family is reported in Table 1.

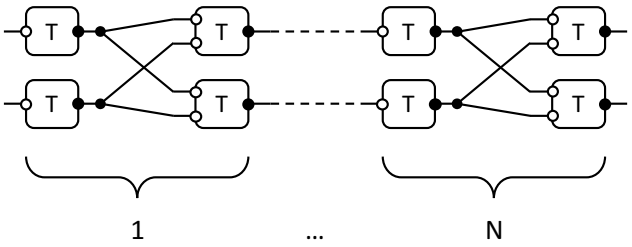


Fig. 16. Rectangular Structures

Table 1. Total Number of Benchmarks

Benchmark Family	**# Instances**	**# State Variables** Boolean	Real	**Uninterpreted Functions**	**MCS**
Linear	$200 * 3$	$8 \sim 2400$	$11 \sim 2403$	$1 \sim 200$	$4 \sim 2394$
Rectangular	$200 * 3$	$32 \sim 9600$	$38 \sim 9606$	$4 \sim 800$	$16 \sim 13197$
Random	$200 * 20$	$8 \sim 2000$	$11 \sim 2000$	$1 \sim 200$	$1 \sim 1800$
Total	5200				

8.3. Results

8.3.1. Linear structures

We first analyzed scalability on linear TMR structures. The results of this comparison are presented in Figure 17: the x axis represents the length of the chain, while on the y axis we report the time needed to compute the minimal cut sets.

The pure SMT generation reaches the timeout for TMR chains with 1 (V111), 2 (V121), and 3 (V123) voters with a length that is respectively 20, 19, and 16 modules. Differently, the predicate abstraction based approach is able to generate the minimal cut sets for a V123 TMR chain of length 200 within 400 seconds. In case of V111 and V121, the computation for all instances terminated successfully under 14 and 65 seconds.

The chains with two and three voters are much harder to deal with, as witnessed by the relative degradation in performance of both techniques. In fact, the presence of additional voters increases the number of fault variables, and the overall number of cut sets. However, the compositional algorithm degrades much less than the basic one.

8.3.2. Rectangular structures

We then analyzed the rectangular structure as in Figure 16. These architectures have the characteristics of extending the linear benchmarks by adding a second dimension to the structure. The base size of a rectangular structure is composed of four TMRs (i.e., modules "T" in Figure 16, each link represents a triple signal), thus the complexity increases much faster compared to a linear structure. This benchmark shows the scalability issues that a pure SMT-based approach experiences when dealing with this structures. In fact, being this benchmark more complex compared to the linear structures, the concrete technique was not able to compute the set of the minimal cut sets for a structure larger than 4 modules, which decreases to 3 in case of 3 voters structures. Differently, the predicate abstraction technique was able to compute the result of a structure with 196 base modules in case of 1 voters TMRs. When considering structures based on 2 and 3 voters, the predicate abstraction completed the analysis for respectively 122 and 61 modules.

8.3.3. Random structures

In order to evaluate the performance of modular abstraction, we built a random generator of DAG-like structures. The problems are generated by picking a module type from the set of possible ones, adding it to the network with inputs selected from inputs of the system or outputs of previously introduced modules, until the target system size is reached. In order to be able to relate numbers of modules and verification complexity, we imposed that maximal diameter of the system of being not greater than $\sqrt{n}$ where n is the number of redundant modules. For the architecture of Figure 3,

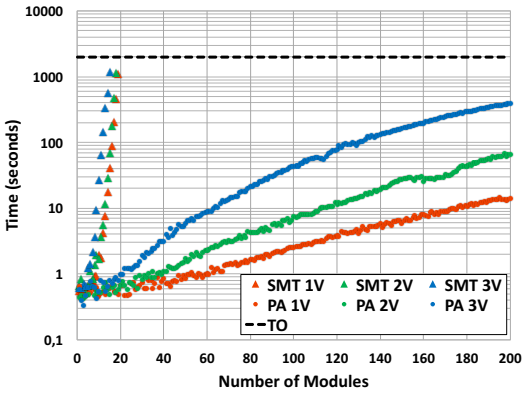


Fig. 17. Linear structures

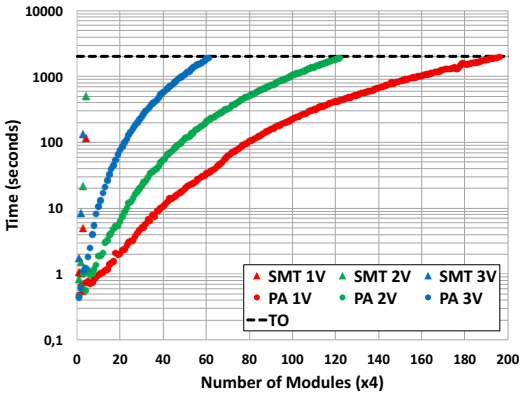


Fig. 18. Rectangular structures

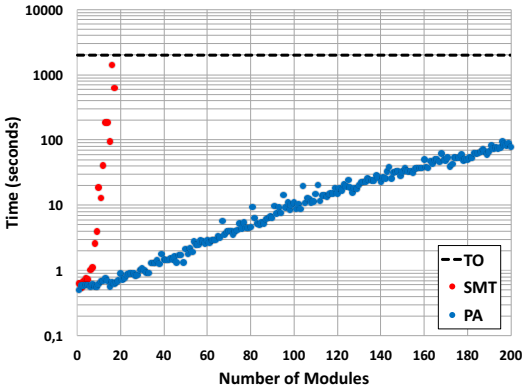


Fig. 19. Random structures

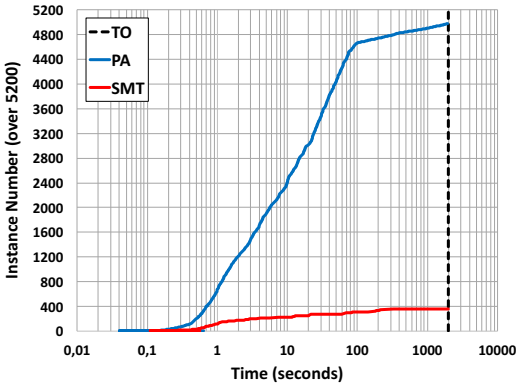


Fig. 20. All instances

used in this paper as a running example [AS74], the modular abstraction technique is able to perform FTA in 0.1 seconds, while the concrete case takes 1.8 seconds. Both methods construct the set of 102 minimal cut sets.

Figure 19 shows the scalability performance on a set of 4000 randomly generated architectures, distributed in 20 instances for each of the 200 possible count of redundant modules. Each point in this chart represents the average time needed to compute the 20 instances of the problem. The results of this test clearly illustrate the improvement of the predicate abstraction technique, which is always able to perform the analysis in less than 100 seconds for each instance, while the SMT-based approach went above the time-out on all architectures with more than 17 modules.

The gain in performance of the predicate abstraction technique can be observed by analyzing the overall results. Figure 20 shows a cactus chart considering all instances described in this section, where a point (x, y) represents the time x required to solve the problem y, by sorting all problems by execution time. Analyzing those results, the advantage of the modular abstraction technique is evident considering that it was able to solve 4979 instances (under the time-out of 2000 seconds), while this number is reduced to 363 in case of the SMT-based approach.

In the monolithic case, the bottleneck is clearly the AllSMT procedure (with optimizations described in [BCT07]), due to the excessive number of cut sets. In the compositional case, the main source of inefficiency is the generation of the BDD. This cost appears to be hard to limit, but we remark that we are obtaining an expensive quantification by partitioning and inlining.

8.4. General remarks

The technique based on Predicate Abstraction demonstrated a significant performance advantage, compared to the AllSMT one. Table 2 provides an overview of the time computation needed by the two techniques to perform the min-

Table 2. Predicate Abstraction vs. AllSMT: required resources

Solved Instances	SMT		PA	
	Timeout	Total Time	Timeout	Total Time
363	2000	17869	0.67	164
4979	NA	NA	2000	287299

Table 3. Predicate Abstraction vs. AllSMT: solved instances

Timeout	Solved Instances		Total Time	Solved Instances	
	SMT	PA		SMT	PA
10^0	118	659	10^2	143	258
10^1	223	2412	10^3	267	1113
10^2	307	4659	10^4	345	2656
10^3	360	4904	10^5	363	4729

imal cut sets computation. In particular, the AllSMT technique required a timeout of 2000 seconds (for each instance) to solve 384 benchmarks, while the approach based on Predicate Abstraction solves the same amount of problems with a timeout of 0.67 seconds, proving a gain in performance of 3 orders of magnitude. This result guarantees the possibility to evaluate significantly more architectural patterns in the same amount of time. Analyzing this numbers from a different perspective, while the SMT-based technique solves 363 instances, the predicate abstraction analyzes more than 1 million (i.e., $(2000/0.67) * 363 = 1083582$) architectures with similar complexity.

Moreover, given a timeout of 10 seconds (for each benchmark) the AllSMT technique is able to analyze 223 architectures, while the modular abstraction approach solved 2412 problems. Similarly, having a total time limit of 100k seconds (i.e., 27 hours) the two approaches would solve 363 and 4729 instances, where the latter refers to the abstraction based technique. Those results are listed in Table 3.

9. Conclusions and Future Work

Redundancy plays a fundamental role in the design of complex critical systems. In this paper we presented a new approach to the automated analysis of redundancy architectures, which is able to produce a symbolic representation of the reliability function from the architectural description.

The process relies on formal methods in different phases: we model the architecture using SMT(EUF) [BSST09]; we adopt a miter construction [Bra93] to describe the occurrence of a top-level failure; we reduce the problem of extracting the set of relevant fault configurations by reduction to an AllSMT problem [LNO06]; predicate abstraction [CCF$^+$07] is at the core of a compositional method for the efficient extraction of the set of cut sets; Binary Decision Diagrams [Bry92] are used to extract a symbolic representation of the reliability function.

The proposed approach allows us to automatically obtain results that previously could be obtained only with manual techniques, and to generalize them by relaxing a number of restrictive hypotheses. The proposed compositional method scales very well, and allows us to analyze configurations of realistic size, composed by thousands of basic blocks. The performance advantage of the abstraction method can be also used to analyze a higher number of different possible designs, thus covering a larger space in the same amount of time.

In the future, we will investigate the following directions. First, we will extend the method to deal with architectures where redundancy is applied to blocks processing real-valued signals. Here, different errors can be classified according to numerically characterized discrepancies (e.g., the distance with respect to the reference signal), and different solutions for voting (e.g., average, median value) are possible. Second, we will investigate the case where faults are associated with dynamics, and the problem is no longer "combinatorial" in nature. Particularly interesting is the extension to a probabilistic analysis based on Markov Decision Processes [BCK$^+$14]. Finally, we will apply the proposed techniques to the problem of the automated exploration of the design space. More specifically, we will integrate the techniques proposed here within an engine to solve multi-valued optimization cost functions, so that the best compromise between cost, weight, power consumption, and reliability can be explored.

References

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

[AL81] Tom Anderson and Peter A Lee. *Fault tolerance, principles and practice*. Prentice/Hall International, 1981.

[AS74] Jacob A. Abraham and Daniel P. Siewiorek. An Algorithm for the Accurate Reliability Evaluation of Triple Modular Redundancy Networks. *IEEE Trans. Computers*, 23(7):682–692, 1974.

[BBC+05] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi A. Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. MathSAT: Tight Integration of SAT and Mathematical Decision Procedures. *J. Autom. Reasoning*, 35(1-3):265–293, 2005.

[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.

[BCF+06] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Alessandro Santuari, and Roberto Sebastiani. To Ackermann-ize or Not to Ackermann-ize? On Efficiently Handling Uninterpreted Function Symbols in *SMT*(*EUF*). In Miki Hermann and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, volume 4246 of *Lecture Notes in Computer Science*, pages 557–571. Springer, 2006.

[BCGM15] Marco Bozzano, Alessandro Cimatti, Alberto Griggio, and Cristian Mattarei. Efficient Anytime Techniques for Model-Based Safety Analysis. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 603–621. Springer, 2015.

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

[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+11] Marco Bozzano, Alessandro Cimatti, Oleg Lisagor, Cristian Mattarei, Sergio Mover, Marco Roveri, and Stefano Tonetta. Symbolic Model Checking and Safety Assessment of Altarica models. *Electronic Communications of the EASST*, 46, 2011.

[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.

[BCT07] Marco Bozzano, Alessandro Cimatti, and Francesco Tapparo. Symbolic Fault Tree Analysis for Reactive Systems. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio 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.

[BGL+00] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saıdi, N. Shankar, and Other Authors. An overview of SAL. In *Proc. of the 5th NASA Langley Formal Methods Workshop*, 2000.

[BLBM07] Christophe Bauer, Kristen Lagadec, Christian Bès, and Marcel Mongeau. Flight control system architecture optimization for fly-by-wire airliners. *Journal of guidance, control, and dynamics*, 30(4):1023–1029, 2007.

[Bra93] Daniel Brand. Verification of large synthesized designs. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7-11, 1993*, pages 534–537, 1993.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

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

[BS97] Glenn Bruns and Ian Sutherland. Model checking and fault tolerance. In *International Conference on Algebraic Methodology and Software Technology*, pages 45–59. Springer, 1997.

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[BV07] Marco Bozzano and Adolfo 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.

[BVÅ+03] M. Bozzano, A. Villafiorita, O. Åkerlund, P. Bieber, C. Bougnol, E. Böde, M. Bretschneider, A. Cavallo, et al. ESACS: an integrated methodology for design and safety analysis of complex systems. *Proc. ESREL 2003*, pages 237–245, 2003. Balkema Publisher.

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014.

[CCF+07] Roberto Cavada, Alessandro Cimatti, Anders Franzén, Krishnamani Kalyanasundaram, Marco Roveri, and R. K. Shyamasundar. Computing Predicate Abstractions by Integrating BDDs and SMT solvers. In *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings*, pages 69–76. IEEE Computer Society, 2007.

[CDT13] Alessandro Cimatti, Michele Dorigatti, and Stefano Tonetta. OCRA: A tool for checking the refinement of temporal contracts. In Ewen Denney, Tevfik Bultan, and Andreas Zeller, editors, *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 702–705. IEEE, 2013.

[Čep11] Marko Čepin. Reliability Block Diagram. In *Assessment of Power System Reliability*, pages 119–123. Springer, 2011.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.

[CMT89] Gianfranco Ciardo, Jogesh K. Muppala, and Kishor S. Trivedi. SPNP: Stochastic Petri Net Package. In *Petri Nets and Performance Models, The Proceedings of the Third International Workshop, PNPM '89, Kyoto, Japan, December 11-13, 1989*, pages 142–151. IEEE Computer Society, 1989.

[Cor64] International Business Machines Corporation. SATURN V – launch vehicle digital computer: Simplex models. Technical Note NASA Part No. 50M35010, NASA, 1964.

[DBL07] *Formal Methods in Computer-Aided Design, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proc.* IEEE Computer Society, 2007.

[DBL09] *Proc. of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*. IEEE, 2009.

[DR01] Yves Dutuit and Antoine Rauzy. New insights into the assessment of k-out-of-n and related systems. *Rel. Eng. & Sys. Safety*, 72(3):303–314, 2001.

[FHT+07] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure. *JSAT*, 1(3-4):209–236, 2007.

[FM04] Michele Favalli and Cecilia Metra. TMR voting in the presence of crosstalk faults at the voter inputs. *IEEE Transactions on Reliability*, 53(3):342–348, 2004.

[GS97] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[HKNP06] Andrew Hinton, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer, 2006.

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[HTK10] Masashi Hamamatsu, Tatsuhiro Tsuchiya, and Tohru Kikuno. On the Reliability of Cascaded TMR Systems. In Yutaka Ishikawa, Dong Tang, and Hiroshi Nakamura, editors, *16th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2010, Tokyo, Japan, December 13-15, 2010*, pages 184–190. IEEE Computer Society, 2010.

[Jan97] Tomasz Janowski. On bisimulation, fault-monotonicity and provable fault-tolerance. In *International Conference on Algebraic Methodology and Software Technology*, pages 292–306. Springer, 1997.

[JH05] Anjali Joshi and Mats Per Erik Heimdahl. Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In Rune Winther, Bjørn Axel Gran, and Gustav 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.

[JS91] Geraint Jones and Mary Sheeran. Relations and Refinement in Circuit Design. In *Proc. BCS FACS Workshop on Refinement, Workshops in Computing*, pages 133–152. Springer-Verlag, 1991.

[JW10] Jonathan M. Johnson and Michael J. Wirthlin. Voter insertion algorithms for FPGA designs using triple modular redundancy. In Peter Y. K. Cheung and John Wawrzynek, editors, *Proceedings of the ACM/SIGDA 18th International Symposium on Field Programmable Gate Arrays, FPGA 2010, Monterey, California, USA, February 21-23, 2010*, pages 249–258. ACM, 2010.

[JHMW06] Anjali Joshi, Mats P.E. Heimdahl, Steven P. Miller, and Mike Whalen. Model-Based Safety Analysis. NASA/CR-2006-213953, 2006.

[KK07] I. Koren and C.M. Krishna. *Fault-Tolerant Systems*. Morgan-Kaufman, 2007.

[KKZ05] Joost-Pieter Katoen, Maneesh Khattri, and Ivan S. Zapreev. A Markov Reward Model Checker. In *Second International Conference on the Quantitative Evaluaiton of Systems (QEST 2005), 19-22 September 2005, Torino, Italy*, pages 243–244. IEEE Computer Society, 2005.

[lay] LayerZero Power Systems, Inc. https://www.layerzero.com/innovations/Industry-Firsts/index.html.

[LJL07] Sungjae Lee, Jae-il Jung, and Inhwan Lee. Voting structures for cascaded triple modular redundant modules. *IEICE Electronic Express*, 4(21):657–664, 2007.

[LNO06] Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. SMT Techniques for Fast Predicate Abstraction. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 424–437. Springer, 2006.

[LQJ] Tan Lanfang, Tan Qingping, and Li Jianli. Specification and Verification of the Triple-Modular Redundancy Fault Tolerant System using CSP. In *Proc. The Fourth International Conference on Dependability (DEPEND)*, pages 14–17. IARIA.

[LS04] Shuvendu K. Lahiri and Sanjit A. Seshia. The UCLID Decision Procedure. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 475–478. Springer, 2004.

[Mat16] Cristian Mattarei. *Scalable Safety and Reliability Analysis via Symbolic Model Checking: Theory and Applications*. PhD thesis, University of Trento, Trento, Italy, 2 2016.

[MBBS15] Anastasia Mavridou, Eduard Baranov, Simon Bliudze, and Joseph Sifakis. Configuration Logics: Modelling Architecture Styles. In Christiano Braga and Peter Csaba Ölveczky, editors, *Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers*, volume 9539 of *Lecture Notes in Computer Science*, pages 256–274. Springer, 2015.

[McM07] Kenneth L. McMillan. Interpolants and Symbolic Model Checking. In Byron Cook and Andreas Podelski, editors, *Verification, Model*

Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings, volume 4349 of *Lecture Notes in Computer Science*, pages 89–90. Springer, 2007.

[Mon93] Giorgio Mongardi. Dependable computing for railway control systems. In Carl E. Landwehr, Brian Randell, and Luca Simoncini, editors, *Dependable Computing for Critical Applications 3*, pages 255–277, Vienna, 1993. Springer.

[RAB+95] R. K. Ranjan, A. Aziz, R. K. Brayton, C. Pixley, and B. Plessier. Efficient bdd algorithms for synthesizing and verifying finite state machines. In *Proceedings of the IEEE/ACM International Workshop on Logic Synthesis (IWLS95)*, Lake Tahoe (NV), 1995.

[Rau93] Antoine Rauzy. New algorithms for fault trees analysis. *Reliability Engineering & System Safety*, 40(3):203–211, 1993.

[Rau01] Antoine Rauzy. Mathematical foundations of minimal cutsets. *IEEE Trans. Reliability*, 50(4):389–396, 2001.

[SIQW95] William H. Sanders, W. Douglas Obal II, Muhammad A. Qureshi, and F. K. Widjanarko. The UltraSAN Modeling Environment. *Perform. Eval.*, 24(1-2):89–115, 1995.

[SLM09] João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-Driven Clause Learning SAT Solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

[Som98] Fabio Somenzi. CUDD: CU decision diagram package release 2.3.0. *University of Colorado at Boulder*, 1998.

[TIC+05] Darshan D. Thaker, Francois Impens, Isaac L. Chuang, Rajeevan Amirtharajah, and Frederic T. Chong. Recursive TMR: Scaling Fault Tolerance in the Nanoscale Era. *IEEE Design & Test of Computers*, 22(4):298–305, 2005.

[Tri02] Kishor S. Trivedi. SHARPE 2002: Symbolic Hierarchical Automated Reliability and Performance Evaluator. In *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*, page 544. IEEE Computer Society, 2002.

[VGRH81] W.E. Vesely, F.F. Goldberg, N.H. Roberts, and D.F. Haasl. Fault Tree Handbook. Technical Report NUREG-0492, Systems and Reliability Research Office of Nuclear Regulatory Research U.S. Nuclear Regulatory Commission, 1981.

[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.

[Yeh96] YC Yeh. Triple-triple redundant 777 primary flight computer. In *Aerospace Applications Conference, 1996. Proc., IEEE*, volume 1, pages 293–307. IEEE, 1996.

[ZLMR09] Miaomiao Zhang, Zhiming Liu, Charles Morisset, and Anders P. Ravn. Design and Verification of Fault-Tolerant Components. In Michael J. Butler, Cliff B. Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, *Methods, Models and Tools for Fault Tolerance*, volume 5454 of *Lecture Notes in Computer Science*, pages 57–84. Springer, 2009.

A. Proofs

Lemma 7.1(Sequential equivalence). Given the combinatorial components M, M_1, and M_2, if $M_1 \equiv M_2$ then $M \triangleright M_1 \equiv M \triangleright M_2$ and $M_1 \triangleright M \equiv M_2 \triangleright M$.

Proof:
if $M_1 = \langle \vec{I}_1, \vec{O}_1, \vec{F}_1, \pi_1 \rangle$, $M_2 = \langle \vec{I}_2, \vec{O}_2, \vec{F}_2, \pi_2 \rangle$, and $M = \langle \vec{I}, \vec{O}, \vec{F}, \pi \rangle$,
then by hypothesis

$\forall \vec{I}_1, \vec{O}_1, \vec{O}_2, \vec{F}_1.($
$(\pi_1(\vec{I}_1, \vec{O}_1, \vec{F}_1) \iff \pi_2(\vec{I}_1, \vec{O}_2, \vec{F}_1)) \wedge (\vec{O}_1 \iff \vec{O}_2)) \wedge$
$\forall \vec{I}, \vec{I}_2, \vec{O}, \vec{F}.($
$(\pi(\vec{I}, \vec{O}, \vec{F}) \iff \pi(\vec{I}_2, \vec{O}, \vec{F})) \wedge (\vec{I} \iff \vec{I}_2))$

iff

$\forall \vec{I}_1, \vec{O}_1, \vec{O}_2, \vec{F}_1, \vec{I}, \vec{I}_2, \vec{O}, \vec{F}.($
$(\pi_1(\vec{I}_1, \vec{O}_1, \vec{F}_1) \iff \pi_2(\vec{I}_1, \vec{O}_2, \vec{F}_1)) \wedge$
$(\pi(\vec{I}, \vec{O}, \vec{F}) \iff \pi(\vec{I}_2, \vec{O}, \vec{F})) \wedge$
$(\vec{O}_1 \iff \vec{O}_2) \wedge (\vec{I} \iff \vec{I}_2))$

implies

$\forall \vec{I}_1, \vec{O}_1, \vec{O}_2, \vec{F}_1, \vec{I}, \vec{I}_2, \vec{O}, \vec{F}.($
$(\pi_1(\vec{I}_1, \vec{O}_1, \vec{F}_1) \iff \pi_2(\vec{I}_1, \vec{O}_2, \vec{F}_1)) \wedge$
$(\pi(\vec{I}, \vec{O}, \vec{F}) \iff \pi(\vec{I}_2, \vec{O}, \vec{F})))$

implies (i.e., $(a \iff b) \wedge (c \iff d) \implies (a \wedge c) \iff (a \wedge d)$)

$\forall \vec{I}_1, \vec{O}_1, \vec{O}_2, \vec{F}_1, \vec{I}, \vec{I}_2, \vec{O}, \vec{F}.($
$(\pi_1(\vec{I}_1, \vec{O}_1, \vec{F}_1) \wedge \pi(\vec{I}, \vec{O}, \vec{F})) \iff$
$(\pi_2(\vec{I}_1, \vec{O}_2, \vec{F}_1) \wedge \pi(\vec{I}_2, \vec{O}, \vec{F})))$
$\implies$
$\forall \vec{I}_1, \vec{O}, \vec{F}, \vec{F}_1.(\exists \vec{O}_1, \vec{O}_2, \vec{I}, \vec{I}_2.($
$(\pi_1(\vec{I}_1, \vec{O}_1, \vec{F}_1) \wedge \pi(\vec{I}, \vec{O}, \vec{F}) \wedge (\vec{O}_1 \iff \vec{I})) \iff$
$(\pi_2(\vec{I}_1, \vec{O}_2, \vec{F}_1) \wedge \pi(\vec{I}_2, \vec{O}, \vec{F})) \wedge (\vec{O}_2 \iff \vec{I}_2)))$

that expresses the system defined as

$M_1 \triangleright M \equiv M_2 \triangleright M$

Similarly for the case

$M \triangleright M_1 \equiv M \triangleright M_2$.

Lemma 7.2(Parallel equivalence). Given the combinatorial components S_1, S_2, S_3, S_4, if $S_1 \equiv S_2 \wedge S_3 \equiv S_4$ then $S_1 | S_3 \equiv S_2 | S_4$.

Proof:
if $S_1 = \langle \vec{I}_1, \vec{O}_1, \vec{F}_1, \pi_1 \rangle$, $S_2 = \langle \vec{I}_2, \vec{O}_2, \vec{F}_2, \pi_2 \rangle$,
$S_3 = \langle \vec{I}_3, \vec{O}_3, \vec{F}_3, \pi_3 \rangle$, $S_4 = \langle \vec{I}_4, \vec{O}_4, \vec{F}_4, \pi_4 \rangle$

then by hypothesis

$$\forall \vec{I'}, \vec{O'}, \vec{F'}.(\\ \pi_1(\vec{I'}, \vec{O'}, \vec{F'}) \iff \pi_2(\vec{I'}, \vec{O'}, \vec{F'})) \wedge \\ \forall \vec{I''}, \vec{O''}, \vec{F''}.(\\ \pi_3(\vec{I''}, \vec{O''}, \vec{F''}) \iff \pi_4(\vec{I''}, \vec{O''}, \vec{F''}))$$

then

$$\forall \vec{I'}, \vec{O'}, \vec{F'}, \vec{I''}, \vec{O''}, \vec{F''}.(\\ \pi_1(\vec{I'}, \vec{O'}, \vec{F'}) \iff \pi_2(\vec{I'}, \vec{O'}, \vec{F'}) \wedge \\ \pi_3(\vec{I''}, \vec{O''}, \vec{F''}) \iff \pi_4(\vec{I''}, \vec{O''}, \vec{F''}))$$

implies

$$\forall \vec{I'}, \vec{O'}, \vec{F'}, \vec{I''}, \vec{O''}, \vec{F''}.(\\ \pi_1(\vec{I'}, \vec{O'}, \vec{F'}) \wedge \pi_3(\vec{I''}, \vec{O''}, \vec{F''}) \iff \\ \pi_2(\vec{I'}, \vec{O'}, \vec{F'}) \wedge \pi_4(\vec{I''}, \vec{O''}, \vec{F''}))$$

that expresses the system defined as

$$S_1|S_3 \equiv S_2|S_4.$$

Lemma 7.3(Inversion equivalence). Given the combinatorial components M_1, M_2, M_3, M_4, if $M_1 \rightsquigarrow M_2$ and $M_3 \rightsquigarrow M_4$ then $(M_1|M_3) \triangleright (M_2|M_4) \equiv (M_1 \triangleright M_2)|(M_3 \triangleright M_4)$.

Proof.

$$(\exists \vec{O_1}, \vec{O_3}, \vec{I_2}, \vec{I_4} : (\pi_1(\vec{I_1}, \vec{O_1}, \vec{F_1}) \wedge \pi_3(\vec{I_3}, \vec{O_3}, \vec{F_3})) \wedge (\pi_2(\vec{I_2}, \vec{O_2}, \vec{F_2}) \wedge \pi_4(\vec{I_4}, \vec{O_4}, \vec{F_4})) \wedge \vec{O_1} = \vec{I_2} \wedge \vec{O_3} = \vec{I_4}) \equiv \\ (\exists \vec{O_1}, \vec{O_3}, \vec{I_2}, \vec{I_4} : (\pi_1(\vec{I_1}, \vec{O_1}, \vec{F_1}) \wedge \pi_3(\vec{I_3}, \vec{O_3}, \vec{F_3})) \wedge (\pi_2(\vec{I_2}, \vec{O_2}, \vec{F_2}) \wedge \pi_4(\vec{I_4}, \vec{O_4}, \vec{F_4})) \wedge \vec{O_1} = \vec{I_2} \wedge \vec{O_3} = \vec{I_4}) \\ \implies \\ (\exists \vec{O_1}, \vec{O_3}, \vec{I_2}, \vec{I_4} : (\pi_1(\vec{I_1}, \vec{O_1}, \vec{F_1}) \wedge \pi_3(\vec{I_3}, \vec{O_3}, \vec{F_3})) \wedge (\pi_2(\vec{I_2}, \vec{O_2}, \vec{F_2}) \wedge \pi_4(\vec{I_4}, \vec{O_4}, \vec{F_4})) \wedge \vec{O_1} = \vec{I_2} \wedge \vec{O_3} = \vec{I_4}) \equiv \\ (\exists \vec{O_1}, \vec{I_2}, \vec{O_3}, \vec{I_4} : (\pi_1(\vec{I_1}, \vec{O_1}, \vec{F_1}) \wedge \pi_2(\vec{I_2}, \vec{O_2}, \vec{F_2}) \wedge \vec{O_1} = \vec{I_2}) \wedge (\pi_3(\vec{I_3}, \vec{O_3}, \vec{F_3}) \wedge \pi_4(\vec{I_4}, \vec{O_4}, \vec{F_4}) \wedge \vec{O_3} = \vec{I_4})) \\ \implies \\ (\exists \vec{O_1}, \vec{O_3}, \vec{I_2}, \vec{I_4} : (\pi_1(\vec{I_1}, \vec{O_1}, \vec{F_1}) \wedge \pi_3(\vec{I_3}, \vec{O_3}, \vec{F_3})) \wedge (\pi_2(\vec{I_2}, \vec{O_2}, \vec{F_2}) \wedge \pi_4(\vec{I_4}, \vec{O_4}, \vec{F_4})) \wedge \vec{O_1} = \vec{I_2} \wedge \vec{O_3} = \vec{I_4}) \equiv \\ (\exists \vec{O_1}, \vec{I_2} : \pi_1(\vec{I_1}, \vec{O_1}, \vec{F_1}) \wedge \pi_2(\vec{I_2}, \vec{O_2}, \vec{F_2}) \wedge \vec{O_1} = \vec{I_2}) \wedge (\exists \vec{O_3}, \vec{I_4} : \pi_3(\vec{I_3}, \vec{O_3}, \vec{F_3}) \wedge \pi_4(\vec{I_4}, \vec{O_4}, \vec{F_4}) \wedge \vec{O_3} = \vec{I_4}) \\ \implies \\ (M_1|M_3) \triangleright (M_2|M_4) \equiv (M_1 \triangleright M_2)|(M_3 \triangleright M_4).$$

Lemma 7.4(Associativity). Given the combinatorial components M_1, M_2, and M_3, if $M_1 \rightsquigarrow M_2$ and $M_2 \rightsquigarrow M_3$ then $(M_1 \triangleright M_2) \triangleright M_3 \equiv M_1 \triangleright (M_2 \triangleright M_3)$ and $(M_1|M_2)|M_3 \equiv M_1|(M_2|M_3)$.

Proof (sequential case).

$$(\exists \vec{O}_1, \vec{I}_2, \vec{O}_2, \vec{I}_3 : \pi_1(\vec{I}_1, \vec{O}_1, \vec{F}_1) \wedge \pi_2(\vec{I}_2, \vec{O}_2, \vec{F}_2) \wedge \pi_3(\vec{I}_3, \vec{O}_3, \vec{F}_3) \wedge \vec{O}_1 = \vec{I}_2 \wedge \vec{O}_2 = \vec{I}_3) \equiv$$
$$(\exists \vec{O}_1, \vec{I}_2, \vec{O}_2, \vec{I}_3 : \pi_1(\vec{I}_1, \vec{O}_1, \vec{F}_1) \wedge \pi_2(\vec{I}_2, \vec{O}_2, \vec{F}_2) \wedge \pi_3(\vec{I}_3, \vec{O}_3, \vec{F}_3) \wedge \vec{O}_1 = \vec{I}_2 \wedge \vec{O}_2 = \vec{I}_3)$$
$$\Longrightarrow$$
$$(\exists \vec{O}_1, \vec{I}_2, \vec{O}_2, \vec{I}_3 : (\pi_1(\vec{I}_1, \vec{O}_1, \vec{F}_1) \wedge \pi_2(\vec{I}_2, \vec{O}_2, \vec{F}_2) \wedge \vec{O}_1 = \vec{I}_2) \wedge \pi_3(\vec{I}_3, \vec{O}_3, \vec{F}_3) \wedge \vec{O}_2 = \vec{I}_3) \equiv$$
$$(\exists \vec{O}_1, \vec{I}_2, \vec{O}_2, \vec{I}_3 : \pi_1(\vec{I}_1, \vec{O}_1, \vec{F}_1) \wedge (\pi_2(\vec{I}_2, \vec{O}_2, \vec{F}_2) \wedge \pi_3(\vec{I}_3, \vec{O}_3, \vec{F}_3) \wedge \vec{O}_2 = \vec{I}_3) \wedge \vec{O}_1 = \vec{I}_2)$$
$$\Longrightarrow$$
$$(\exists \vec{O}_2, \vec{I}_3 : (\exists \vec{O}_1, \vec{I}_2 : \pi_1(\vec{I}_1, \vec{O}_1, \vec{F}_1) \wedge \pi_2(\vec{I}_2, \vec{O}_2, \vec{F}_2) \wedge \vec{O}_1 = \vec{I}_2) \wedge \pi_3(\vec{I}_3, \vec{O}_3, \vec{F}_3) \wedge \vec{O}_2 = \vec{I}_3) \equiv$$
$$(\exists \vec{O}_1, \vec{I}_2 : \pi_1(\vec{I}_1, \vec{O}_1, \vec{F}_1) \wedge (\exists \vec{O}_2, \vec{I}_3 : \pi_2(\vec{I}_2, \vec{O}_2, \vec{F}_2) \wedge \pi_3(\vec{I}_3, \vec{O}_3, \vec{F}_3) \wedge \vec{O}_2 = \vec{I}_3) \wedge \vec{O}_1 = \vec{I}_2)$$
$$\Longrightarrow$$
$$(M_1 \triangleright M_2) \triangleright M_3 \equiv M_1 \triangleright (M_2 \triangleright M_3).$$

Proof (parallel case).

$$(\pi_1(\vec{I}_1, \vec{O}_1, \vec{F}_1) \wedge \pi_2(\vec{I}_2, \vec{O}_2, \vec{F}_2)) \wedge \pi_3(\vec{I}_3, \vec{O}_3, \vec{F}_3) \equiv$$
$$\pi_1(\vec{I}_1, \vec{O}_1, \vec{F}_1) \wedge (\pi_2(\vec{I}_2, \vec{O}_2, \vec{F}_2) \wedge \pi_3(\vec{I}_3, \vec{O}_3, \vec{F}_3))$$
$$\Longrightarrow$$
$$(M_1|M_2)|M_3 \equiv M_1|(M_2|M_3).$$

```
function prob = abraham(C2M2, C2M3, C2V1, C2V3, C2V2, C6M2, C3V2, C3V1, C6M1, C5M2, C5M3,
      C4M2, C4M3, C4M1, C3M3, C3M2, C3M1, C3V3, C5V1, C5V2, C5V3, C4V1, C4V2, C1V1, C1V2,
      C1M2, C1M3, C6V1, C6V2, C6V3, C1M1, C1V3, C4V3, C6M3, C5M1, C2M1)
    EXPR0000 = (1 - C6V1) * (C6V2 * C6V3) + C6V1 * ((1 - C6V2) * C6V3 + C6V2);
    EXPR0001 = (1 - C1M1) * EXPR0000 + C1M1;
    EXPR0002 = (1 - C1M2) * ((1 - C1M3) * EXPR0000 + C1M3 * EXPR0001) + C1M2 * ((1 - C1M3)
       * EXPR0001 + C1M3);
    EXPR0003 = (1 - C1V3) * EXPR0002 + C1V3;
    EXPR0008 = (1 - C1V2) * EXPR0003 + C1V2;
    EXPR0004 = (1 - C1V1) * ((1 - C1V2) * EXPR0002 + C1V2 * EXPR0003) + C1V1 * EXPR0008;
    EXPR0012 = (1 - C1V1) * EXPR0003 + C1V1;
    EXPR0016 = (1 - C1V1) * ((1 - C1V2) * EXPR0002 + C1V2) + C1V1;
    EXPR0021 = (1 - C1V1) * EXPR0008 + C1V1;
    EXPR0005 = (1 - C4V3) * ((1 - C6M3) * EXPR0004 + C6M3) + C4V3;
    EXPR0009 = (1 - C4V3) * ((1 - C6M3) * EXPR0008 + C6M3) + C4V3;
    EXPR0013 = (1 - C4V3) * ((1 - C6M3) * EXPR0012 + C6M3) + C4V3;
    EXPR0017 = (1 - C4V3) * ((1 - C6M3) * EXPR0016 + C6M3) + C4V3;
    EXPR0022 = (1 - C4V3) * ((1 - C6M3) * EXPR0021 + C6M3) + C4V3;
    EXPR0006 = (1 - C4V2) * EXPR0005 + C4V2;
    EXPR0010 = (1 - C4V2) * EXPR0009 + C4V2;
    EXPR0014 = (1 - C4V2) * EXPR0013 + C4V2;
    EXPR0018 = (1 - C4V2) * EXPR0017 + C4V2;
    EXPR0023 = (1 - C4V2) * EXPR0022 + C4V2;
    EXPR0007 = (1 - C4V1) * ((1 - C4V2) * EXPR0004 + C4V2) + C4V1;
    EXPR0011 = (1 - C4V1) * ((1 - C4V2) * EXPR0008 + C4V2) + C4V1;
    EXPR0015 = (1 - C4V1) * ((1 - C4V2) * EXPR0012 + C4V2) + C4V1;
    EXPR0019 = (1 - C4V1) * ((1 - C4V2) * EXPR0016 + C4V2) + C4V1;
    EXPR0024 = (1 - C4V1) * ((1 - C4V2) * EXPR0021 + C4V2) + C4V1;
    EXPR0050 = (1 - C5V3) * ((1 - C4V1) * EXPR0005 + C4V1) + C5V3;
    EXPR0051 = (1 - C5V3) * ((1 - C4V1) * EXPR0009 + C4V1) + C5V3;
    EXPR0052 = (1 - C5V3) * ((1 - C4V1) * EXPR0013 + C4V1) + C5V3;
    EXPR0053 = (1 - C5V3) * ((1 - C4V1) * EXPR0017 + C4V1) + C5V3;
    EXPR0055 = (1 - C5V3) * ((1 - C4V1) * EXPR0022 + C4V1) + C5V3;
    EXPR0034 = (1 - C5V2) * ((1 - C5V3) * EXPR0006 + C5V3) + C5V2;
    EXPR0035 = (1 - C5V2) * ((1 - C5V3) * EXPR0010 + C5V3) + C5V2;
    EXPR0036 = (1 - C5V2) * ((1 - C5V3) * EXPR0014 + C5V3) + C5V2;
    EXPR0037 = (1 - C5V2) * ((1 - C5V3) * EXPR0018 + C5V3) + C5V2;
    EXPR0039 = (1 - C5V2) * ((1 - C5V3) * EXPR0023 + C5V3) + C5V2;
    EXPR0020 = (1 - C3V3) * ((1 - C5V1) * ((1 - C5V2) * ((1 - C5V3) * ((1 - C4V1) * ((1 -
      C4V2) * EXPR0008 + C4V2 * EXPR0009) + C4V1 * EXPR0010) + C5V3 * EXPR0011) + C5V2 *
      EXPR0051) + C5V1 * EXPR0035) + C3V3 * ((1 - C5V1) * ((1 - C5V2) * EXPR0011 + C5V2) +
      C5V1);
    EXPR0026 = (1 - C3V3) * ((1 - C5V1) * ((1 - C5V2) * ((1 - C5V3) * ((1 - C4V1) * ((1 -
      C4V2) * EXPR0016 + C4V2 * EXPR0017) + C4V1 * EXPR0018) + C5V3 * EXPR0019) + C5V2 *
      EXPR0053) + C5V1 * EXPR0037) + C3V3 * ((1 - C5V1) * ((1 - C5V2) * EXPR0019 + C5V2) +
      C5V1);
    EXPR0027 = (1 - C3V3) * ((1 - C5V1) * ((1 - C5V2) * ((1 - C5V3) * ((1 - C4V1) * ((1 -
      C4V2) * EXPR0021 + C4V2 * EXPR0022) + C4V1 * EXPR0023) + C5V3 * EXPR0024) + C5V2 *
      EXPR0055) + C5V1 * EXPR0039) + C3V3 * ((1 - C5V1) * ((1 - C5V2) * EXPR0024 + C5V2) +
      C5V1);
    EXPR0029 = (1 - C3V3) * ((1 - C5V1) * ((1 - C5V2) * ((1 - C5V3) * ((1 - C4V1) * ((1 -
      C4V2) * EXPR0012 + C4V2 * EXPR0013) + C4V1 * EXPR0014) + C5V3 * EXPR0015) + C5V2 *
      EXPR0052) + C5V1 * EXPR0036) + C3V3 * ((1 - C5V1) * ((1 - C5V2) * EXPR0015 + C5V2) +
      C5V1);
    EXPR0038 = (1 - C3V3) * EXPR0035 + C3V3;
```

B. Matlab function example

Figure 21 shows the reliability function of the architecture described in Figure 3. The terms `CiMj` represent the failure of the module `j` inside the component `i`, and similarly for `CiVj` where `Vj` represents the voter `j`.

```
41      EXPR0041 = (1 - C3V3) * EXPR0037 + C3V3;
        EXPR0042 = (1 - C3V3) * EXPR0039 + C3V3;
        EXPR0044 = (1 - C3V3) * EXPR0036 + C3V3;
        EXPR0054 = (1 - C3V3) * ((1 - C5V1) * EXPR0051 + C5V1) + C3V3;
45      EXPR0057 = (1 - C3V3) * ((1 - C5V1) * EXPR0053 + C5V1) + C3V3;
        EXPR0058 = (1 - C3V3) * ((1 - C5V1) * EXPR0055 + C5V1) + C3V3;
        EXPR0060 = (1 - C3V3) * ((1 - C5V1) * EXPR0052 + C5V1) + C3V3;
        EXPR0025 = (1 - C3M1) * EXPR0027 + C3M1;
        EXPR0030 = (1 - C3M1) * EXPR0029 + C3M1;
50      EXPR0040 = (1 - C3M1) * EXPR0042 + C3M1;
        EXPR0045 = (1 - C3M1) * EXPR0044 + C3M1;
        EXPR0056 = (1 - C3M1) * EXPR0058 + C3M1;
        EXPR0061 = (1 - C3M1) * EXPR0060 + C3M1;
        EXPR0028 = (1 - C3M2) * ((1 - C3M1) * EXPR0026 + C3M1) + C3M2;
55      EXPR0031 = (1 - C3M2) * EXPR0025 + C3M2;
        EXPR0043 = (1 - C3M2) * ((1 - C3M1) * EXPR0041 + C3M1) + C3M2;
        EXPR0046 = (1 - C3M2) * EXPR0040 + C3M2;
        EXPR0059 = (1 - C3M2) * ((1 - C3M1) * EXPR0057 + C3M1) + C3M2;
        EXPR0062 = (1 - C3M2) * EXPR0056 + C3M2;
60      EXPR0076 = (1 - C3M3) * ((1 - C3M2) * EXPR0020 + C3M2 * EXPR0025) + C3M3 * EXPR0031;
        EXPR0077 = (1 - C3M3) * ((1 - C3M2) * EXPR0038 + C3M2 * EXPR0040) + C3M3 * EXPR0046;
        EXPR0079 = (1 - C3M3) * ((1 - C3M2) * EXPR0054 + C3M2 * EXPR0056) + C3M3 * EXPR0062;
        EXPR0071 = (1 - C4M1) * ((1 - C3M3) * ((1 - C3M2) * ((1 - C3M1) * EXPR0026 + C3M1 *
          EXPR0027) + C3M2 * EXPR0025) + C3M3 * EXPR0028) + C4M1;
        EXPR0072 = (1 - C4M1) * ((1 - C3M3) * ((1 - C3M2) * ((1 - C3M1) * EXPR0041 + C3M1 *
          EXPR0042) + C3M2 * EXPR0040) + C3M3 * EXPR0043) + C4M1;
65      EXPR0074 = (1 - C4M1) * ((1 - C3M3) * ((1 - C3M2) * ((1 - C3M1) * EXPR0057 + C3M1 *
          EXPR0058) + C3M2 * EXPR0056) + C3M3 * EXPR0059) + C4M1;
        EXPR0066 = (1 - C4M3) * ((1 - C4M1) * ((1 - C3M3) * ((1 - C3M2) * ((1 - C3M1) *
          EXPR0029 + C3M1 * EXPR0027) + C3M2 * EXPR0030) + C3M3 * EXPR0031) + C4M1) + C4M3;
        EXPR0067 = (1 - C4M3) * ((1 - C4M1) * ((1 - C3M3) * ((1 - C3M2) * ((1 - C3M1) *
          EXPR0044 + C3M1 * EXPR0042) + C3M2 * EXPR0045) + C3M3 * EXPR0046) + C4M1) + C4M3;
        EXPR0069 = (1 - C4M3) * ((1 - C4M1) * ((1 - C3M3) * ((1 - C3M2) * ((1 - C3M1) *
          EXPR0060 + C3M1 * EXPR0058) + C3M2 * EXPR0061) + C3M3 * EXPR0062) + C4M1) + C4M3;
        EXPR0032 = (1 - C4M2) * ((1 - C4M3) * ((1 - C4M1) * ((1 - C3M3) * ((1 - C3M2) * ((1 -
          C3M1) * ((1 - C3V3) * ((1 - C5V1) * ((1 - C5V2) * ((1 - C5V3) * ((1 - C4V1) * ((1 -
          C4V2) * EXPR0004 + C4V2 * EXPR0005) + C4V1 * EXPR0006) + C5V3 * EXPR0007) + C5V2 *
          EXPR0050) + C5V1 * EXPR0034) + C3V3 * ((1 - C5V1) * ((1 - C5V2) * EXPR0007 + C5V2) +
          C5V1)) + C3M1 * EXPR0020) + C3M2 * EXPR0030) + C3M3 * EXPR0028) + C4M1 * EXPR0076) +
          C4M3 * EXPR0071) + C4M2 * EXPR0066;
70      EXPR0047 = (1 - C4M2) * ((1 - C4M3) * ((1 - C4M1) * ((1 - C3M3) * ((1 - C3M2) * ((1 -
          C3M1) * ((1 - C3V3) * EXPR0034 + C3V3) + C3M1 * EXPR0038) + C3M2 * EXPR0045) + C3M3 *
           EXPR0043) + C4M1 * EXPR0077) + C4M3 * EXPR0072) + C4M2 * EXPR0067;
        EXPR0063 = (1 - C4M2) * ((1 - C4M3) * ((1 - C4M1) * ((1 - C3M3) * ((1 - C3M2) * ((1 -
          C3M1) * ((1 - C3V3) * ((1 - C5V1) * EXPR0050 + C5V1) + C3V3) + C3M1 * EXPR0054) +
          C3M2 * EXPR0061) + C3M3 * EXPR0059) + C4M1 * EXPR0079) + C4M3 * EXPR0074) + C4M2 *
          EXPR0069;
        EXPR0033 = (1 - C5M1) * EXPR0032 + C5M1;
        EXPR0048 = (1 - C5M1) * EXPR0047 + C5M1;
        EXPR0064 = (1 - C5M1) * EXPR0063 + C5M1;
75      EXPR0068 = (1 - C5M3) * ((1 - C5M1) * EXPR0067 + C5M1) + C5M3;
        EXPR0049 = (1 - C5M2) * ((1 - C5M3) * EXPR0047 + C5M3 * EXPR0048) + C5M2 * ((1 - C5M3)
          * EXPR0048 + C5M3);
        EXPR0073 = (1 - C5M2) * ((1 - C5M1) * ((1 - C4M2) * EXPR0072 + C4M2) + C5M1) + C5M2;
        EXPR0078 = (1 - C5M2) * ((1 - C5M3) * ((1 - C4M2) * ((1 - C4M3) * EXPR0077 + C4M3) +
          C4M2) + C5M3) + C5M2;
        EXPR0065 = (1 - C3V1) * ((1 - C6M1) * ((1 - C5M2) * ((1 - C5M3) * EXPR0063 + C5M3 *
          EXPR0064) + C5M2 * ((1 - C5M3) * EXPR0064 + C5M3)) + C6M1) + C3V1;
80      EXPR0070 = (1 - C3V1) * ((1 - C6M1) * ((1 - C5M3) * ((1 - C5M1) * EXPR0069 + C5M1) +
          C5M3) + C6M1) + C3V1;



81      EXPR0075 = (1 - C3V1) * ((1 - C6M1) * ((1 - C5M2) * ((1 - C5M1) * ((1 - C4M2) *
          EXPR0074 + C4M2) + C5M1) + C5M2) + C6M1) + C3V1;
        EXPR0080 = (1 - C3V1) * ((1 - C6M1) * ((1 - C5M2) * ((1 - C5M3) * ((1 - C4M2) * ((1 -
          C4M3) * EXPR0079 + C4M3) + C4M2) + C5M3) + C5M2) + C6M1) + C3V1;
        EXPR0081 = (1 - C2V1) * ((1 - C2V3) * ((1 - C2V2) * ((1 - C6M2) * ((1 - C3V2) * ((1 -
          C3V1) * ((1 - C6M1) * ((1 - C5M2) * ((1 - C5M3) * EXPR0032 + C5M3 * EXPR0033) + C5M2
          * ((1 - C5M3) * EXPR0033 + C5M3)) + C6M1 * EXPR0049) + C3V1 * EXPR0049) + C3V2 *
          EXPR0065) + C6M2 * EXPR0065) + C2V2 * ((1 - C6M2) * ((1 - C3V2) * ((1 - C3V1) * ((1 -
           C6M1) * ((1 - C5M3) * ((1 - C5M1) * EXPR0066 + C5M1) + C5M3) + C6M1 * EXPR0068) +
          C3V1 * EXPR0068) + C3V2 * EXPR0070) + C6M2 * EXPR0070)) + C2V3 * ((1 - C2V2) * ((1 -
          C6M2) * ((1 - C3V2) * ((1 - C3V1) * ((1 - C6M1) * ((1 - C5M2) * ((1 - C5M1) * ((1 -
          C4M2) * EXPR0071 + C4M2) + C5M1) + C5M2) + C6M1 * EXPR0073) + C3V1 * EXPR0073) + C3V2
           * EXPR0075) + C6M2 * EXPR0075) + C2V2)) + C2V1 * ((1 - C2V3) * ((1 - C2V2) * ((1 -
          C6M2) * ((1 - C3V2) * ((1 - C3V1) * ((1 - C6M1) * ((1 - C5M2) * ((1 - C5M3) * ((1 -
          C4M2) * ((1 - C4M3) * EXPR0076 + C4M3) + C4M2) + C5M3) + C5M2) + C6M1 * EXPR0078) +
          C3V1 * EXPR0078) + C3V2 * EXPR0080) + C6M2 * EXPR0080) + C2V2) + C2V3);
        EXPR0082 = (1 - C2M1) * EXPR0081 + C2M1;
85      prob = (1 - C2M2) * ((1 - C2M3) * EXPR0081 + C2M3 * EXPR0082) + C2M2 * ((1 - C2M3) *
          EXPR0082 + C2M3);
```

Fig. 21. Reliability function of the architecture in Figure 3