Efficient Analysis of Reliability Architectures via Predicate Abstraction

Marco Bozzano, Alessandro Cimatti, and Cristian Mattarei

Fondazione Bruno Kessler, Trento, Italy,
{bozzano,cimatti,mattarei}@fbk.eu

Abstract. The overall safety of critical systems is often based on the use of redundant architectural patterns, such as Triple Modular Redundancy. Certification procedures in various application domains require an explicit evaluation of the reliability, and the production of various artifacts. Particularly interesting are Fault Trees (FT), that represent in a compact form all the combinations of (basic) faults that are required to cause a (system-level) failure. Yet, such activities are essentially based on manual analysis, and are thus time consuming and error prone.
A recently proposed approach opens the way to the automated analysis of reliability architectures. The approach is based on the use of Satisfiability Modulo Theories (SMT), using the theory of Equality and Uninterpreted Functions ($\mathcal{EUF}$) to represent block diagrams. Within this framework, the construction of FTs is based on the existential quantification of an $\mathcal{EUF}$ formula. Unfortunately, the off-the-shelf application of available techniques, based on the translation into an AllSMT problem, suffers from severe scalability issues.
In this paper, we propose a compositional method to solve this problem, based on the use of predicate abstraction. We prove that our method is sound and complete for a wide class of system architectures. The presented approach greatly improves the overall scalability with respect to the monolithic case, obtaining speed-ups of various orders of magnitude. In practice, this approach allows for the verification of architectures of realistic systems.

Keywords: Formal Verification,Reliability Architectures,Fault Tree Analysis,Satisfiability Modulo Theory,Redundant Systems

1 Introduction

Redundancy is a well known solution used in the design of critical system. In order to increase the dependability of a system, components carrying out important functions are replicated, and their effects combined by means of dedicated modules such as voters. A typical schema is Triple Module Redundancy (TMR) where three components are connected by a voter. This solution can be instantiated multiple times within the same system, in cascading stages organized in different structures [3,37,28].

The reliability analysis for such architectures is based on the construction of so-called Fault Trees [45]. A Fault Tree (FT) identifies all the configurations of faults that can lead to an undesired event (e.g., loss of a system function). The construction of FT's from a model are in general not carried out automatically, and are thus costly, tedious and error prone. A recent exception is the work in [13], where the problem of analyzing reliability architectures is cast in the framework of Satisfiability Modulo Theories (SMT) [6]. Functional blocks are represented in the theory of Equality and Uninterpreted Functions ($\mathcal{EUF}$) as uninterpreted functions. Redundancy is modeled by the repetition of the same function block, combined with blocks representing the voting mechanisms. The possible occurrence of faults is modeled by the introduction of Boolean fault variables. Within this framework, FTs are directly generated by the collection of values to the fault variables that make an $\mathcal{EUF}$ formula satisfiable. In fact, the construction of such FTs is a variation of the AllSMT problem [35] where the assignments to the fault variables are required to be minimal with respect to set inclusion. Unfortunately, the techniques based on [35] can be seen as a monolithic enumeration of the disjucts of the DNF of the resulting formula, are are often subject to a blow up. This prevents the construction of FT (and ultimately the reliability analysis) for systems of realistic size.

In this paper, we propose a new method for the compositional computation of FTs for the analysis of redundancy architectures. The key technical insight is the use of predicate abstraction to partition the construction of FT. More specifically, the computation of the FT for a DAG of concrete components proceeds in two steps: first, we combine the abstraction of the individual components under a suitable set of predicates, carrying out an SMT-based quantification, thus obtaining a pure Boolean model; then, we compute the FT for such model using BDD-based projection techniques [14]. We prove that the approach is sound, i.e. the FTs computed on the abstract system are the same as the ones computed directly on the original, concrete system.

The approach was implemented within the NuSMV3 system, on top of the MathSAT5 [21] SMT solver, and we carried also out an experimental evaluation to test the scalability. On small-sized examples, where the monolithic approach requires already a significant computation time, the new method performs orders of magnitude better. Even more important, the new method scales dramatically better, and is able to generate fault trees with hundreds of blocks in less than one minute. The increased capacity allowed us to analyze some classical architectures (e.g. [3,46,37,28]) that are out of reach for the previous technique [13].

The paper is structured as follows. In Section 2 we discuss some relevant related work. In Section 3 we present some logical background. In Section 4 we define the problem, and discuss the limitations of the previous solutions. In Section 5 we present our approach. In Section 6 we formally define the approach and prove its soundness. In Section 7 we describe the experimental evaluation. In Section 8 we draw some conclusions, and discuss future work.

2 Related Work

In recent years, there has been a growing interest in techniques for model-based safety assessment [33]. The perspective of model-based safety assessment 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 model-based techniques allows safety analysis to be more tractable in terms of time consumption and costs. Such techniques must be able to verify functional correctness and assess system behavior in presence of faults [17,4,11,16].

A key difference with respect to our approach is that these techniques focus on the analysis of the *behavior* of dynamical systems, whereas our approach aims at evaluating characteristics of redundancy architectures, independently of components' behavior. Our approach builds upon the work in [13], which is based on the calculus of Equality and Uninterpreted Functions ($\mathcal{EUF}$), and makes use of Satisfiability Modulo Theory (SMT) techniques for verification [7,27].

The techniques based on Markov Decision Process and Probabilistic Petri Nets [34,29,44,19,40] are widely used in industry for the quantitative evaluation and reliability analysis. However, such approaches are not able to provide a uniform and completely automated process, and in fact, the link between the reliability evaluation and the qualitative safety assessment analysis is performed manually. Thus, this is a key difference between the approach proposed in [13] and the current techniques for the analysis of reliability architectures.

In this work we rely on *NuSMV3*, that is a complete verification and validation framework for model based analysis. *NuSMV3* is based on an open source verification engine [20], that supports BDD-based and SAT-based finite state model checking. At its core, *NuSMV3* uses the SMT solver MathSAT [10,21], that supports several theories like linear arithmetic over reals and integers, difference logic, bit vectors, uninterpreted functions, and equality. In addition to verification, *NuSMV3* also provides complex capabilities to perform safety assessment, in particular, FTA [14] and reliability evaluation.

3 Background

Traditionally, dynamical systems are modeled as finite state systems: their state can be represented by means of assignments to a specified set of variables [30]. In symbolic model checking, they are represented by means of Boolean logic, where (Boolean) variables are combined together via Boolean connectives (e.g. conjunction, disjunction, negation). In this approach, sets of states are represented by the Boolean formula corresponding to the characteristic function of the set. The symbolic analyses of dynamic systems, most notably symbolic model checking techniques (e.g. [39,8,38]) rely on efficient ways to represent and manipulate Boolean formulae, in particular Binary Decision Diagrams [18], and, more recently, Boolean satisfiability (SAT) solvers [41].

Boolean logic, however, is a rather limited representation, and fails to represent many important classes of systems. This limitation has been lifted with the advent of Satisfiability Modulo Theory (SMT) [6], where the formula is not pure Boolean, but it is expressed in some background theory such as Real and Integer Arithmetic ($\mathcal{LA}(\mathbb{Q})/\mathcal{LA}(\mathbb{Z})$), Difference Logic ($\mathcal{DL}$), and Bit Vectors ($\mathcal{BV}$). On top of SMT solver there are many different verification algorithms that can be used [25,43,22,23]. In this paper we will focus primarily on the theory of Equality and Uninterpreted Functions ($\mathcal{EUF}$), where variables range over an unspecified domain, and function symbols can be declared, but have no specific property, except for the fact that they are functions, i.e. $(x = y) \rightarrow (f(x) = f(y))$. Moreover, we use predicate abstraction in order to approximate a concrete system using a set of formulas (predicates). Our approach makes use of an AllSMT procedure [35] that efficiently implements predicate abstraction by enumerating all the satisfying assignments over the set of predicates using an SMT solver.

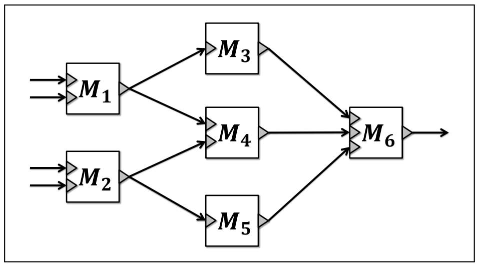
The target of our approach is to improve the analysis of reliability architectures, and in particular the techniques for Model-Based Safety Assessment such as the construction of Fault Trees and Failure Mode and Effects Analysis (FMEA) tables, which can be performed automatically by reduction to symbolic model checking [16,11,15,14,12].

4 The problem: analysis of reliability architectures

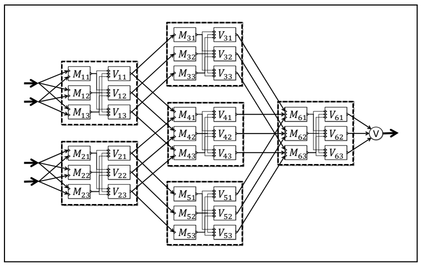
The evaluation of architectural patterns is an essential ingredient for the development of safety critical system, due to the fact that such systems have to guarantee an high reliability. When a specific component is essential to guarantee correct and safe operation of the system, a standard practice in safety engineering is to encapsulate it in a redundant architectural pattern. This practice aims at increasing the reliability of the redunded component.

One of the most well-known architectural pattern is the Triple Modular Redundancy (TMR) architecture [3,5,26,42,31]. It consists in triplicating the module that is critical for the reliability of the system, and feeding one voter with their outputs. Specifically, considering each redundant module as a functional component, receiving an input and providing an output, the voter returns the value computed by the majority of the redunded modules. This approach allows us to varying, and hopefully increase, the reliability of the system, which depends on the reliability of each voter and each component, in addition to the displacement and connections between them. The analysis of reliability architectures, in general, is performed manually, due to the lack of specific techniques addressing both modeling and verification. The manual approach is supported by several specific algorithms [28,36] that can aid the verification and analysis of reliability of TMR chains. However, such approaches cannot be generalized in order to cover a full set of architectural patterns.

Recent studies on the verification of architectural patterns [13] aim at automating the analysis of reliability architectures. The idea proposed in [13] consists in defining the behavior of components using uninterpreted functions. Such formalism has the capability to describe the functional behavior of the components without giving any details of their implementation. Uninterpreted func-

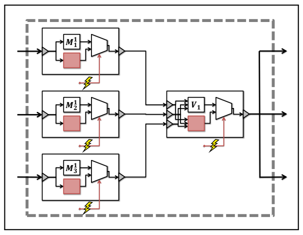


(a) Nominal System

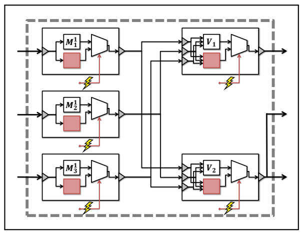

M_{11} M_{12} M_{13} V_{11} V_{12} V_{13}
M_{21} M_{22} M_{23} V_{21} V_{22} V_{23}
M_{31} M_{32} M_{33} V_{31} V_{32} V_{33}
M_{41} M_{42} M_{43} V_{41} V_{42} V_{43}
M_{51} M_{52} M_{53} V_{51} V_{52} V_{53}
M_{61} M_{62} M_{63} V_{61} V_{62} V_{63}
V

(b) Redundant System

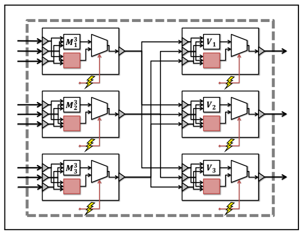
Fig. 1: Network of combinatorial components [3]



(a) 1 port, 1 voter



(b) 1 port, 2 voters



(c) 3 ports, 3 voters

Fig. 2: Modular redundancy examples

tions have no specific properties, except that they have to provide the same outputs when given the same inputs. Moreover, faulty behavior can be modeled simply by leaving the output of a faulty component unconstrained.

5 The approach

In this work we concentrate on the equivalence checking between nominal and faulty systems with redundant modules. The idea is to provide the same inputs to both structures and evaluate under which conditions the outputs are different. This evaluation relies on Model-Based Fault Tree Analysis [14], which consists of generating all the faults configurations such that it is possible to reach an undesirable behavior (a.k.a. Top Level Event). A faults configuration, also called cut-set, is minimal if it is not possible to reach the TLE by removing a fault from this set, and in this work we concentrate on the minimal cut-sets generation.

Figure 1 shows a graphical representation of the nominal (1a) and redundant (1b) configurations of the network example introduced in [3]. Each redundant module, as shown in Figure 2, is then extended to take into account faults, namely by placing the nominal (the M and V modules) and faulty behaviors (the light red modules) in parallel. Figure 2 illustrates some examples of this

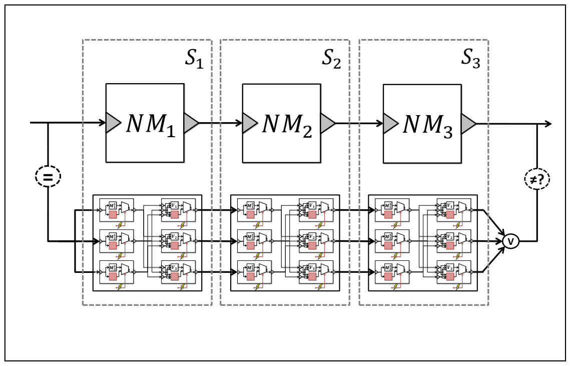


Fig. 3: Comparison TMRs and faultless modules

extension, for different architectures. The selection between nominal and faulty behaviors is realized by a multiplexer that receives a fault event as input. Finally, Figure 3 illustrates our approach to system equivalence, in the case of linear architectures. It consists in equating the output of the nominal architecture with the output of the extended redundant architecture. The same approach can be generalized to the case of Tree or DAG structures, their evaluation being similar to the linear ones.

In this work, we refer to modules that integrate both nominal and redundant system definition for each stage. This approach allows us to keep aligned (w.r.t architectural patterns) nominal and redundant systems, by construction. The idea is to define each module, composed of nominal and redundant behavior, using an abstract definition that preserves their behavior, while permitting a significant improvement of the performances of the routines that analyze them.

The predicate abstraction is defined for each individual redundant component, and formalized in Equation 1. Specifically, given a component defined as an SMT formula $\Gamma_{\mathbb{B}\cup\mathbb{D}}(I, O, F)$ over input and output ports, and faults events, we want to define a Boolean formula $\Psi_{\mathbb{B}}(A_I, A_O, F)$ over input and output predicates (defined as ϕ_I and ϕ_O, then bound to A_I and A_O) by performing a quantifier elimination over concrete input and output ports (the sets I and O), and fault events. The relation between these predicates and concrete ports is defined by

$$\begin{aligned}&\Psi_{\mathbb{B}}(A_I, A_O, F) = \exists I, O.(\Gamma_{\mathbb{B}\cup\mathbb{D}}(I, O, F) \wedge \\ &A_I \iff \phi_I(I) \wedge A_O \iff \phi_O(O))\end{aligned} \tag{1}$$

From the Boolean formula representing an abstract component, it is possible to generate an SMV module that encodes it. On top of this, we can encode a network composed of individual abstract components. In Section 6 we show that this network is equivalent to the abstraction of a network composed of concrete components, and use this result for FTA.

Given that abstract formulas are Boolean, we can analyze them using a BDD-based engine. Moreover, the network definition that we introduced in this paper allows for the generation of an optimal variable ordering that guarantees high verification performances. Equation 2 describes a system as the composition of different modules; in particular, it represents the configuration shown in Figure 1. This notation, similar to the relational language introduced in [32], consists of defining two operators: sequential composition ($\triangleright$) and parallel composition ($|$). The former relates components that are connected in a sequential fashion, linking outputs of the first component with inputs of the second one. Parallel composition, on the other hand, juxtaposes the set of ports from different components, which run in parallel.

$$(M_1|M_2) \triangleright (D|D) \triangleright (M_3|M_4|M_5) \triangleright M_6 \tag{2}$$

The framework described in this paper enables the definition of any tree- or DAG-shaped structure. Three special combinatorial components can be used to connect inputs and outputs of different components, in order to implement: duplication of values (module D), simple propagation of input values (I module, a.k.a. identity) and arbitrary reconfiguration of signals (R module). For instance, in Equation 2 we use D modules in order to duplicate outputs of the M_1 and M_2 components.

6 Abstraction

In this section we formally define our approach, based on predicate abstraction, and we show that it allows for an efficiently generation of FTs. We concentrate on networks of combinatorial components used to define TMR architectures. A combinatorial component, according to Definition 1, is a system with input and output ports, a set of faults signals and a formula. Intuitively, such components do not have time evolution (i.e., they are combinatorial) and the values of the output ports are computed only over current inputs and faults.

Definition 1 (Combinatorial component). *A combinatorial component is a tuple $S = \langle \boldsymbol{P}, F, \pi \rangle$, where:*

- $\boldsymbol{P} = \boldsymbol{P_O} \| \boldsymbol{P_I}$ *are the terms representing vector ports, sequentially split into input and output (i.e. the symbol $\|$ defines vectors concatenation). Each port can have Boolean ($\mathbb{B}$) or Data ($\mathbb{D}$) type, while faults are only Boolean;*
- F *is the set of faults events;*
- $\pi(\boldsymbol{P_I}, \boldsymbol{P_O}, F)$ *is an SMT formula over ports and faults, where each term belongs to $\mathbb{B}$ or $\mathbb{D}$.*

We also define two special combinatorial components whose purpose is to formalize the abstraction. Specifically, the *abstractor component* (compare Definition 2) is used to translate a set of concrete values into their abstract counterpart, whereas the *concretizer component* (Definition 3) generates instances of concrete values satisfying the input predicates.

Definition 2 (Abstractor combinatorial component). *A combinatorial component $A = \langle \boldsymbol{P}, F, \alpha \rangle$ is called abstractor if:*

– $F = \emptyset$;
– $\boldsymbol{P} = \boldsymbol{P_I} \| \boldsymbol{P_O}$;
– $\boldsymbol{P_I}$ *is the vector of input ports belonging to* $\mathbb{D}$;
– $\boldsymbol{P_O}$ *is the vector of output ports belonging to* $\mathbb{B}$;
– $\alpha(\boldsymbol{P_I}, \boldsymbol{P_O}, \emptyset)$ *is an SMT formula over input and output ports.*

Definition 3 (Concretizer combinatorial component). *A combinatorial component* $C = \langle \boldsymbol{P}, F, \gamma \rangle$ *is called concretizer if:*

– $F = \emptyset$;
– $\boldsymbol{P} = \boldsymbol{P_I} \| \boldsymbol{P_O}$;
– $\boldsymbol{P_I}$ *is the vector of input ports belonging to* $\mathbb{B}$;
– $\boldsymbol{P_O}$ *is the vector of output ports belonging to* $\mathbb{D}$;
– $\gamma(\boldsymbol{P_I}, \boldsymbol{P_O}, \emptyset)$ *is an SMT formula over input and output ports.*

Definition 4 formalizes the sequential composition of two components S' and S''. The idea is to connect the output ports of S' to the input ports of S''. The resulting component S has the same input ports as S', the same output ports of S'' and the union of the faults of S' and S''. Concretizer and abstractor components allow us to express the abstraction of module S as the sequential composition $C \triangleright S \triangleright A$.

Definition 4 (Sequential composition). *Given two combinatorial components* $S' = \langle \boldsymbol{P'}, F', \pi' \rangle$ *and* $S'' = \langle \boldsymbol{P''}, F'', \pi'' \rangle$, *such that* $|\boldsymbol{P'_O}| = |\boldsymbol{P''_I}|$, *the sequential composition* $S = \langle \boldsymbol{P}, F, \pi \rangle$, *denoted* $S = S' \triangleright S''$ *is defined by:*

– $\boldsymbol{P_I} = \boldsymbol{P'_I}$;
– $\boldsymbol{P_O} = \boldsymbol{P''_O}$;
– $F = F' \cup F''$;
– $\pi(\boldsymbol{P_I}, \boldsymbol{P_O}, F) = \exists \boldsymbol{P'_O}, \boldsymbol{P''_I} . \pi'(\boldsymbol{P'_I}, \boldsymbol{P'_O}, F') \wedge$
 $\pi''(\boldsymbol{P''_I}, \boldsymbol{P''_O}, F'') \wedge \boldsymbol{P'_O} = \boldsymbol{P''_I}$.

Similarly, parallel composition of two components is defined as follows.

Definition 5 (Parallel composition). *Given two combinatorial components* $S' = \langle \boldsymbol{P'}, F', \pi' \rangle$ *and* $S'' = \langle \boldsymbol{P''}, F'', \pi'' \rangle$, *such that* $F' \cap F'' = \emptyset$, *the parallel composition* $S = \langle \boldsymbol{P}, F, \pi \rangle$, *denoted* $S = S' | S''$, *is defined by:*

– $\boldsymbol{P_I} = \boldsymbol{P'_I} \| \boldsymbol{P''_I}$;
– $\boldsymbol{P_O} = \boldsymbol{P'_O} \| \boldsymbol{P''_O}$;
– $F = F' \cup F''$;
– $\pi(\boldsymbol{P_I}, \boldsymbol{P_O}, F) = \pi'(\boldsymbol{P'_I}, \boldsymbol{P'_O}, F') \wedge \pi''(\boldsymbol{P''_I}, \boldsymbol{P''_O}, F'')$.

Definition 6 expresses the equivalence between combinatorial components. Intuitively, two combinatorial components are equivalent if their relational formulas have the same value for each assignment to input and output ports, and faults.

Definition 6 (System equivalence). *Given two combinatorial components $S' = \langle \boldsymbol{P'}, F', \pi' \rangle$ and $S'' = \langle \boldsymbol{P''}, F'', \pi'' \rangle$, such that $F' = F''$ and $\boldsymbol{P'} = \boldsymbol{P''}$, they are called system equivalent, denoted $S' \equiv S''$, if and only if*
$\forall M = \langle p_{I1}, ..., p_{In}, p_{O1}, ..., p_{Om}, f_1, ..., f_{Ik} \rangle : \pi'(M) \iff \pi''(M)$.

In this work we concentrate on Fault Tree Analysis [45], and specifically on the generation of the Minimal Cut-Sets (MCSs) as formally defined in 8. This analysis provides a subset of the cut-sets (see Definition 7), which represents all fault configurations such that there exists an assignment to input and output ports making a specific event, called top level event (TLE), true. In this case, we consider to have two different systems, nominal and redundant, and the TLE is the predicate representing the difference between the outputs, by providing to them the same input.

Definition 7 (Cut-Sets). *Given a combinatorial component $S = \langle \boldsymbol{P}, F, \pi \rangle$ and a predicate $\mathcal{T}(\boldsymbol{P_O})$, called Top Level Event; the set of cut-sets, denoted CS, is defined as follows:*

$$CS(S, \mathcal{T}) = \{f \in 2^F | \exists p_I \in 2^{\boldsymbol{P_I}}, p_O \in 2^{\boldsymbol{P_O}} . \pi(p_I, p_O, f) \wedge \mathcal{T}(p_O) = \top\}$$

A minimal cut-set is defined as follows, by keeping only cut-sets that are minimal fault configurations.

Definition 8 (Minimal Cut-Sets). *Given a combinatorial component $S = \langle \boldsymbol{P}, F, \pi \rangle$ and a predicate $\mathcal{T}(\boldsymbol{P_O})$, the set of minimal cut-sets, denoted MCS, is defined as follows:*

$$MCS(S, \mathcal{T}) = \{cs \in CS(S, \mathcal{T}) | \forall cs' \in CS(S, \mathcal{T}),\ cs' \subseteq cs \implies cs' = cs\}$$

6.1 Modular abstraction equivalence

In this work we evaluate redundant networks by using modular predicate abstraction. In order to show the soundness of our approach, we prove that, given a system composed of concrete modules, it is possible to substitute each individual module with its abstract counterpart. This result is stated in Theorem 1. We organize the proof using the following lemmas. Lemma 1 states that the if two combinatorial components are equivalent, it is possible to sequentially combine them with a third component and preserve the equivalence. Lemma 2 states a similar result for parallel composition.

Lemma 1 (Reduction equivalence). *Given the combinatorial components S, S', and S'', if $S' \equiv S''$ then $S \triangleright S' \equiv S \triangleright S''$ and $S' \triangleright S \equiv S'' \triangleright S$.*

Lemma 2 (Parallel equivalence). *Given the combinatorial components S'_1, S''_1, S'_2, S''_2, if $S'_1 \equiv S''_1 \wedge S'_2 \equiv S''_2$ then $S'_1 | S'_2 \equiv S''_1 | S''_2$.*

Theorem 1 allows us to generate an equivalent network of combinatorial components by using only abstract 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.

Theorem 1 (Modular abstraction equivalence). *Given a combinatorial component* $S = S_1 \triangleright \ldots \triangleright S_n$*, a set of abstractors* $A_1, A_2, ..., A_n$ *and a set of concretizers* $C_1, C_2, ..., C_n$*, where* $\mathcal{C}(S) = C_1 \triangleright S_1 \triangleright S_2 \triangleright \ldots \triangleright S_{n-1} \triangleright S_n \triangleright A_n$ *and* $\mathcal{A}(S) = C_1 \triangleright S_1 \triangleright A_1 \triangleright \ldots \triangleright C_n \triangleright S_n \triangleright A_n$*, such that* $\forall i \in \{1, ..., n\}.\ |\boldsymbol{P_O^{C_i}}| = |\boldsymbol{P_I^{S_i}}| \wedge |\boldsymbol{P_O^{S_i}}| = |\boldsymbol{P_I^{A_i}}|$

if $$\forall i \in \{2, ..., n\}. A_{i-1} \triangleright C_i \triangleright S_i \triangleright A_i \equiv S_i \triangleright A_i$$

then $$\mathcal{C}(S) \equiv \mathcal{A}(S)$$

Proof. by hypothesis $$S_n \triangleright A_n \equiv A_{n-1} \triangleright C_n \triangleright S_n \triangleright A_n$$

then (by Lemma 1) $$\mathcal{C}(S) \equiv (C_1 \triangleright S_1 \triangleright S_2 \triangleright ... \triangleright S_{n-1}) \triangleright (S_n \triangleright A_n) \equiv (C_1 \triangleright S_1 \triangleright S_2 \triangleright ... \triangleright S_{n-1}) \triangleright (A_{n-1} \triangleright C_n \triangleright S_n \triangleright A_n)$$

then (by hypothesis) $$S_{n-1} \triangleright A_{n-1} \equiv A_{n-2} \triangleright C_{n-1} \triangleright S_{n-1} \triangleright A_{n-1}$$

then (by Lemma 1) $$... \triangleright S_{n-2}) \triangleright (S_{n-1} \triangleright A_{n-1}) \triangleright (C_n \triangleright S_n \triangleright A_n) \equiv ... \triangleright S_{n-2}) \triangleright (A_{n-2} \triangleright C_{n-1} \triangleright S_{n-1} \triangleright A_{n-1}) \triangleright (C_n \triangleright S_n \triangleright A_n)$$

then, keep applying hypothesis and Lemma 1 it is possible to conclude that
$$(C_1 \triangleright S_1) \triangleright (S_2 \triangleright A_2) \triangleright (C_3 \triangleright ... \equiv (C_1 \triangleright S_1) \triangleright (A_1 \triangleright C_2 \triangleright S_2 \triangleright A_2) \triangleright (C_3 \triangleright ... \equiv \mathcal{A}(S)$$

The results stated in Theorem 1 is very general; it can be applied to different abstractions, provided that the hypothesis of the theorem holds. In the case of stages that are a parallel composition of modules, the hypothesis can be proved with Lemma 2, and this is an important aspect when dealing with Tree and DAG systems. As a corollary, we obtain that it is possible to compute the MCSs for the concrete system on the abstract system.

Corollary 1 (Computation of Minimal Cut-Sets). *If a combinatorial component* $S = S_1 \triangleright \ldots \triangleright S_n$*, the abstractors* $A_1, ..., A_n$*, and the concretizers* $C_1, ..., C_n$ *satisfy the hypothesis of Theorem 1, then* $MCS(\mathcal{C}(S), \mathcal{T}) = MCS(\mathcal{A}(S), \mathcal{T})$*.*

7 Experiments

7.1 Implementation

We implemented our approach on top of the NuSMV3 system, a verification tool built on top of NuSMV2 [20] and MathSAT [21]. NuSMV3 provides various SMT-based verification algorithms, and various engines for predicate abstraction [1,2]. The functionalities that are relevant for this paper are the ability to deal with $\mathcal{EUF}$ theory, predicate abstraction via AllSMT [35], and the capability to generate Fault Trees with probabilistic evaluations as described in [13].

Our implementation takes a description of a nominal model, its counterpart expressed with redundancy schemas, and can generate either the monolithic problem or the compositional problem, where the various components are modeled with fault variables and predicates describing discrepancies between the nominal and redundant flow.

We instantiated the framework described in Section 5 using the following abstraction, which expresses, given a set of input and output ports, the equivalence between nominal values and their extended version. More precisely, considering a stage with a nominal component having i_n, o_n as input and output ports, and a redundant module duplicating the signals with i_1, i_2, o_1, o_2 as ports, our abstraction generates the predicates $\{(i_n = i_1), (i_n = i_2)\}$ as input, and $\{(o_n = o_1), (o_n = o_2)\}$ as output.

In order to use the results of Section 6, we have to prove that the hypothesis of Theorem 1 holds for our predicates. For this purpose, we carried out an equivalence checking using the MathSAT SMT solver. Specifically, we proved that the formula $\nexists M : \neg(\pi_\alpha(M) \iff \pi_\gamma(M))$ is unsatisfiable for each SMV module implementation, where π_α and π_γ represent, respectively, abstract and concrete formula modulo predicates, as expressed in Theorem 1. Thus, each sequence $C_i \triangleright S_i \triangleright A_i$ explicitly represents an abstract component, and it is used as a single module that is computed using AllSMT-based predicate abstraction techniques.

The generation of Fault Trees, in the form of Binary Decision Diagrams [18], provided 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 2, 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 setting for the experimental evaluation comprises the generation of the abstract modules, for each of the possible pair of nominal and redundant components represented in Figure 2, and then caching their machine representation. The time needed to perform such process is not taken into account in the scalability evaluation, however this operation takes on average 5 seconds with a maximum time of 10 seconds. The target of our evaluation consists in Fault Tree Analysis (generation of MCSs), with a top level event stating that the output of the nominal network differs from the redundant one. The library of abstract components consists of 12 different redundancy configurations with 1, 2 and 3 voters per stage. The system configuration for the standard methodology of [13], without predicate abstraction, is similar to the setting with modular abstraction with the difference that each module is a concrete representation with real variables and $\mathcal{EUF}$ functions. The algorithms used in both cases are based on Fault Tree generation as proposed in [14]; given the difference between concrete and abstract, in the first case we use SMT-based techniques, whereas for the latter we use the BDD-based ones.

7.2 Experimental evaluation

We compared the performance of the monolithic and compositional approaches on a wide set of benchmarks, including randomly generated and real-world architectures. Whenever both techniques terminated, we checked the correctness by comparing the Fault Trees. We ran the experiments on an Intel Xeon E3-1270 at 3.40GHz, with a timeout of 1000 seconds, and a memory limit of 1 GB.

Linear Structures We first analyzed the scalability of the approach on linear TMR structures. The TMR chains experiments consider networks of length n with 1, 2 and 3 voters, with different combinations of structures. The results of this comparison are presented in Figure 4: the x axis represents the length of the chain, while on the y axis there is the time needed to compute the minimal cut-sets. The concrete generation reaches the timeout starting from a TMR chain with 1 and 2 voters of length 20, while with 3 voters, it is not able to evaluate more than 10 stages within the timeout. The modular abstraction approach is able to perform FTA in less than 110 seconds for a TMR chain of length 140, both with 1, 2 and 3 voters.

The two and three voters schemas are much harder to deal with (as witnessed by the relative degrade 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. In the case of compositional, partitioning helps to limit the impact on performance. However, the compositional approach is vastly superior to the monolithic one which shows a significant degrade in performance.

Scalability on Tree and DAG structures We then analyzed tree and DAG diagrams, first considering the design description presented in [3], that describes a DAG redundant structure as shown in Figure 1. In this case, the modular abstraction technique is able to perform FTA in 0.025 seconds, while the concrete case takes 4.5 seconds. Both methods construct the set of 102 minimal cut-sets.

The analysis of a real-word system architecture concerned the verification of the redundancy management of the Boeing 777 Primary Flight Computation, as described in [46]. The model considers a system with 36 redundant modules and 123 possible faults. In this case, the technique based on predicate abstraction takes 1.07 seconds to generate the Fault Tree composed of 195 minimal cut-sets. Differently, the monolithic approach takes 4680 seconds (1 hour and 18 minutes).

In order to evaluate the performance of modular abstraction, we built a random generator of Tree and DAG 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 the increase of system diameter between two consecutive layers is at most two modules. This means that a random tree structure with length 140 has a maximum diameter of 22 modules (i.e. max diameter with n modules is $2 * \sqrt{n} - 1$).

The set of possible components is defined with modules with 1, 2, and 3 inputs and a single output, in addition to the special components D, which replicates the input to two equal set of outputs, and an identity module I.

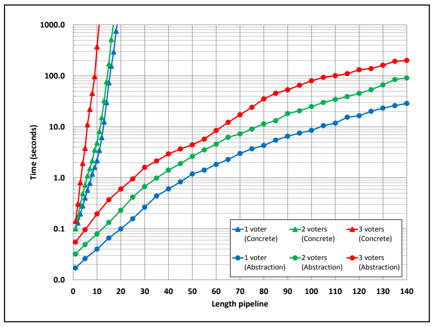


Fig. 4: Scalability evaluation on linear structures

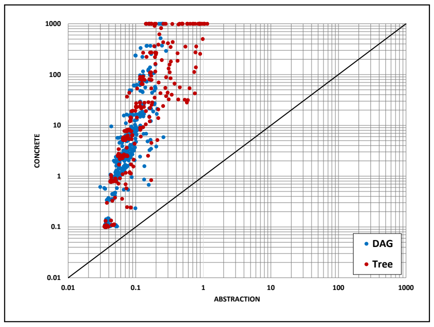


Fig. 5: Tree (Red) and DAG (Blue) comparison

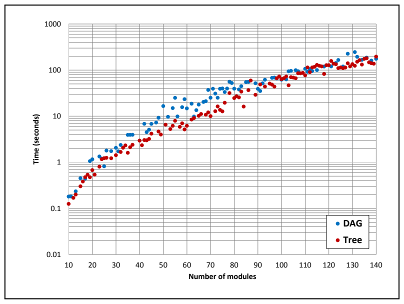


Fig. 6: Tree and DAG scalability: abstraction

The random generation of Tree and DAG networks allows us to compare the performances of two approaches. Figure 5 shows a scatter plot of the results for networks of size until 25, with red and blue points representing respectively Tree and DAG architectures. The results of this test clearly illustrate the improvement due to the abstraction, which is able to perform the analysis in less than 1.5 seconds for each instance, with an average gain in performance that is in the order of 10^2 (i.e. Gain (Min, Avg, Max) $= (2, 6 * 10^2, 7 * 10^3)$).

The scalability evaluation of the modular approach in the case of Tree and DAG structure is shown in Figure 6. In this chart, the x axis represents the number of modules composing the network, while the y axis shown the total time to compute the full set of minimal cut-sets. The module count in the case of DAG does not consider the components of type D or I, due to the fact that they essentially express links between stages. The results shows that the performance in the case of Linear, Tree or DAG structure are almost comparable, in fact almost all the time is spent on the BDD quantification of predicates.

In the monolithic case, the bottleneck is clearly the AllSMT procedure (with optimizations described in [14]), due to the excessive number of cut-sets. In the compositional case, the time for initializing the library accounts in total for less than 1 minute. This cost is payed only once, and the necessary abstractions can be cached. Once the library is initialized, the main source of inefficiency is the generation of the BDD. This cost appears hard to limit, but we remark that we are obtaining an expensive quantification by partitioning and inlining.

8 Conclusion

In this paper we tackled the problem of automated safety assessment of redundancy architectures. In this work, we enhance the approach proposed in [13], where functional blocks are modeled within the SMT($\mathcal{EUF}$) framework. We focus on the construction of Fault Trees, that is a fundamental step in [13]: this step was tackled as a problem of AllSMT [35] and turned out to be a bottleneck. Here we propose a compositional technique for the construction of fault trees that relies on the idea of predicate abstraction, and partitions the problem, trading one large quantifier-elimination operation with several (but much simpler) operations. We prove the correctness of the decomposition, and provide an implementation realized on top the MathSAT5 solver. An experimental evaluation demonstrates dramatic improvements in terms of scalability with respect to the monolithic quantification. This makes it possible to construct Fault Trees with more than 400 minimal cut-sets from 2^{6*140} (10^{250}) possible fault configurations. The availability of this tool allows us to automatically obtain results for realistic configurations that were previously out of reach.

In the future, we will investigate the integration of these techniques into an architecture decomposition framework, based on contract-based design [24]. We will also analyze the problem of synthesizing the best configuration for a given cost function. Future work will also consider the analysis of various forms of deployment, where functions are run on the same platform. This form of analysis, also known as Common Cause Analysis, can be expressed in the modeling framework, but it is currently unclear if the compositional analysis is retained.

References

1. *Formal Methods in Computer-Aided Design, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proc.* IEEE Computer Society, 2007.
2. *Proc. of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*. IEEE, 2009.
3. J.A. Abraham and D.P. Siewiorek. An algorithm for the accurate reliability evaluation of triple modular redundancy networks. *IEEE Trans. on Comp.*, 100(7):682–692, 1974.
4. O Akerlund, P Bieber, E Boede, M Bozzano, M Bretschneider, C Castel, A Cavallo, M Cifaldi, J Gauthier, A Griffault, et al. ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects. *Proc. ERTS*, 2006, 2006.
5. Tom Anderson and Peter A Lee. *Fault tolerance, principles and practice.* Prentice/Hall International, 1981.
6. Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Biere et al. [9], pages 825–885.

7. S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saıdi, N. Shankar, et al. An overview of sal. In *Proc. of the 5th NASA Langley Formal Methods Workshop*, 2000.
8. A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.
9. A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *FAIA*. IOS Press, 2009.
10. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. MathSAT: Tight Integration of SAT and Mathematical Decision Procedures. *Journal of Automated Reasoning*, 35:265–293, 2005.
11. M. Bozzano, A. Cimatti, J.P. Katoen, V.Y. Nguyen, T. Noll, and M. Roveri. Safety, dependability, and performance analysis of extended AADL models. *The Computer Journal*, doi: 10.1093/com, March 2010.
12. M. Bozzano, A. Cimatti, O. Lisagor, C. Mattarei, S. Mover, M. Roveri, and S. Tonetta. Symbolic model checking and safety assessment of altarica models. *ECEASST*, 46, 2012.
13. M. Bozzano, A. Cimatti, and C. Mattarei. Automated analysis of reliability architectures. In *ICECCS*, pages 198–207. IEEE Computer Society, 2013.
14. M. Bozzano, A. Cimatti, and F. Tapparo. Symbolic Fault Tree Analysis for Reactive Systems. In *Proc. Symposium on Automated Technology for Verification and Analysis (ATVA 2007)*, pages 162–176, 2007.
15. M. Bozzano and A. Villafiorita. The FSAP/NuSMV-SA Safety Analysis Platform. *International Journal on Software Tools for Technology Transfer*, 9(1):5–24, 2007.
16. M. Bozzano and A. Villafiorita. *Design and Safety Assessment of Critical Systems.* CRC Press (Taylor and Francis), an Auerbach Book, 2010.
17. 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.
18. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
19. Gianfranco Ciardo, Jogesh Muppala, and Kishor Trivedi. SPNP: stochastic Petri net package. In *Petri Nets and Performance Models, 1989. PNPM89., Proc. of the Third International Workshop on*, pages 142–151. IEEE, 1989.
20. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV : a new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):410–425, March 2000.
21. Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.
22. Alessandro Cimatti, Sergio Mover, and Stefano Tonetta. SMT-Based Verification of Hybrid Systems. In Jörg Hoffmann and Bart Selman, editors, *AAAI*, 2012.
23. Alessandro Cimatti, Sergio Mover, and Stefano Tonetta. SMT-based scenario verification for hybrid systems. *Formal Methods in System Design*, 42(1):46–66, 2013.
24. Alessandro Cimatti and Stefano Tonetta. A property-based proof system for contract-based design. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pages 21–28. IEEE, 2012.
25. L. de Moura, S. Owre, H. Rueß, J. M. Rushby, N. Shankar, M. Sorea, and A. Tiwari. Sal 2. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500. Springer, 2004.
26. Michele Favalli and Cecilia Metra. TMR voting in the presence of crosstalk faults at the voter inputs. *Reliability, IEEE Transactions on*, 53(3):342–348, 2004.

27. M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *J. on Satisfiability, Boolean Modeling and Computation*, 1(3-4):209–236, 2007.
28. Masashi Hamamatsu, Tatsuhiro Tsuchiya, and Tohru Kikuno. On the reliability of cascaded TMR systems. In *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on*, pages 184–190. IEEE, 2010.
29. Andrew Hinton, Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: A tool for automatic verification of probabilistic systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 441–444. Springer, 2006.
30. Gerard J Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.
31. J.M. Johnson and M.J. Wirthlin. Voter insertion algorithms for fpga designs using triple modular redundancy. In *Proc. of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 249–258. ACM, 2010.
32. Geraint Jones and Mary Sheeran. Relations and refinement in circuit design. In *3rd Refinement Workshop*, volume 90, pages 133–152. Citeseer, 1990.
33. Anjali Joshi, Mike Whalen, and Mats P.E. Heimdahl. Modelbased safety analysis: Final report. Technical report, 2005.
34. J-P Katoen, Maneesh Khattri, and IS Zapreevt. A markov reward model checker. In *Quantitative Evaluation of Systems, 2005. Second International Conference on the*, pages 243–244. IEEE, 2005.
35. Shuvendu K. Lahiri, Robert Nieuwenhuis, and Albert Oliveras. SMT Techniques for Fast Predicate Abstraction. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 424–437. Springer, 2006.
36. Tan Lanfang, Tan Qingping, and Li Jianli. Specification and verification of the triple-modular redundancy fault tolerant system using csp. In *DEPEND 2011, The Fourth International Conference on Dependability*, pages 14–17, 2011.
37. Sungjae Lee, Jae il Jung, and Inhwan Lee. Voting structures for cascaded triple modular redundant modules. *Ieice Electronic Express*, 4(21):657–664, 2007.
38. Kenneth L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.
39. K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
40. William H. Sanders, W Douglas Obal II, Muhammad A. Qureshi, and FK Widjanarko. The ultrasan modeling environment. *Perf. Evaluation*, 24(1):89–115, 1995.
41. João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In Biere et al. [9], pages 131–153.
42. Darshan D Thaker, Rajeevan Amirtharajah, F Impens, IL Chuang, and Frederic T Chong. Recursive TMR: Scaling fault tolerance in the nanoscale era. *Design & Test of Computers, IEEE*, 22(4):298–305, 2005.
43. Stefano Tonetta. Abstract model checking without computing the abstraction. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 2009.
44. Kishor S Trivedi. Sharpe 2002: Symbolic hierarchical automated reliability and performance evaluator. In *Dependable Systems and Networks, 2002. DSN 2002. Proc.. International Conference on*, page 544. IEEE, 2002.
45. W.E. Vesely, M. Stamatelatos, J. Dugan, J. Fragola, J. Minarick III, and J. Railsback. Fault Tree Handbook with Aerospace Applications, 2002.
46. YC Yeh. Triple-triple redundant 777 primary flight computer. In *Aerospace Applications Conference, 1996. Proc., IEEE*, volume 1, pages 293–307. IEEE, 1996.