UNIVERSITÀ DEGLI STUDI
DI TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE
ICT International Doctoral School

Scalable Safety and Reliability Analysis via Symbolic Model Checking: Theory and Applications

Cristian Mattarei

Advisor
Alessandro Cimatti
Head of Embedded Systems Unit,
Fondazione Bruno Kessler

Co-Advisor
Marco Bozzano
Senior Researcher,
Fondazione Bruno Kessler

February 2016

UNIVERSITÀ DEGLI STUDI
DI TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE

ICT International Doctoral School

PhD Committee

David Jones
The Boeing Company

Luigi Portinale
University of Piemonte Orientale

Roberto Sebastiani
University of Trento

Christel Seguin
ONERA

Joseph Sifakis
EPFL

Abstract

Assuring safety and reliability is fundamental when developing a safety critical system. Road, naval and avionic transportation; water and gas distribution; nuclear, eolic, and photovoltaic energy production are only some examples where it is mandatory to guarantee those properties. The continuous increasing in the design complexity of safety critical system calls for a never ending sought of new and more advanced analytical techniques. In fact, they are required to assure that undesired consequences are highly improbable.

In this Thesis we introduce a novel methodology able to raise the bar in the area of automated safety and reliability analysis. The proposed approach integrates a series of techniques, based on symbolic model checking, into the current development process of safety critical systems. Moreover, our methodology and the resulting techniques are thereafter applied to a series of real-world case studies, developed in collaboration with authoritative entities such as NASA and the Boeing Company.

Keywords

[Model-Based Safety Assessment, Symbolic Model Checking, Safety Assessment, Reliability Analysis, Fault Tree Analysis, Contract-Based Design, Minimal Cutsets]

Contents

List of Tables

List of Figures

List of Acronyms

ATC	Air Traffic Control
BDD	Binary Decision Diagram
CBD	Contract-Based Design
CBSA	Contract-Based Safety Assessment
CD&R	Conflict Detection and Resolution
CS	Cutset
CTL	Computational Tree Logic
DAG	Directed Acyclic Graph
DMR	Double Modular Redundancy
FBW	Fly-by-Wire
FDIR	Fault Detection Identification and Recovery
FMEA	Fault Mode and Effects Analysis
FTA	Fault Tree Analysis
GSEP	Ground Separated Aircraft
IC3	Incremental Construction of Inductive Clauses for Indubitable Correctness
LTL	Linear Temporal Logic
MBSA	Model-Based Safety Assessment
MEA	More Electric Airplane

MCS Minimal Cutsets

PDR Property Directed Reachability

SMT Satisfiability Modulo Theory

SSEP Self Separated Aircraft

TMR Triple Modular Redundancy

V&V Validation and Verification

WBS Wheel Braking System

1

Introduction

If a machine is expected to be infallible,
it cannot also be intelligent.

– Alan Turing

It is the 1906, and Lee de Forest invents the vacuum tube and makes way for the active electronics. 30 years later, this result had a huge impact in WWII which pushed on this technology and initiated the electronic revolution. Colossus, the world's first programmable computer, was one of the most important application of such technology, but in this Thesis we refer to this period for a different reason: the emerging of safety and reliability engineering.

Electronic researches during WWII contributed in the development of technological applications such as radio, radar, and television. At the same time, the vacuum tubes were also the main cause of equipment failure, in fact they required to be replaced five times more often then all other equipments. This recurring issue required to investigate on the definition of specific analysis, able to attribute the cause of such unreliability of the electrical components. In a general perspective, the term reliability attributes to the system capability of behaving in accordance with its prescribed functionality, in fact a failure of a vacuum tube in an electrical

device can cause the entire system not to working property. Differently, system safety is the property of not causing damage, risk, or injury. After WWII, specific studies in this direction arose from the necessity to deal with the increasing level of complexity in military aircraft and ballistic missile systems.

Over the years, the vacuum tubes were replaced by transistors, and their successive miniaturization has allowed for the increasing in system capability and complexity. In parallel to this trend, safety and reliability engineering have had to evolve by introducing new and more efficient approaches able to support the design, and avoid unintended behaviors, of such complex systems. In the current era, the problem of assuring safety and reliability affects the design of systems that are definitely more pervasive than the purely military ones. Most notably areas of application for such disciplines are road, naval and avionic transportation; water and gas distribution; and nuclear, eolic, and photovoltaic energy production. Guaranteeing safety and reliability in these applications is mandatory, thus they are categorized as *safety critical systems*. The process that guides the development of a safety critical system is highly controlled and standardized by the competent authorities. In fact, releasing a certificate of system conformance requires to guarantee that system requirements, defined at the early stages of the development, are fairly derived into the system and sub-systems design, correctly implemented into the production phase, and - finally - that the concrete system implementation is in accordance with sub-systems, system, and the original requirements definition. Each of these phases is characterized by a set of well established analysis and methodology, which guides the system design through an incremental refinement from initial requirements definition to the final system implementation. The resulting process has two parallel flows: one that analyzes the system under normal conditions, and the other that evaluates its robustness in

presence of components' failure. The former is the system development *V-Model*, and the latter is called *safety assessment*.

Modern safety critical systems have become so complex that their safety cannot be shown solely by test, and whose logic is nearly impossible to comprehend without the aid of analytical tools. The approach that, in the last decades, emerged to cope with such complexity is the use of formal methods. In practice, a system behavior can be defined with a variety of diagrams, textual descriptions, and operational procedures, but in all cases they must be well defined and tailored to avoid ambiguous interpretations. The application of formal methods solves this issue by providing a set of mathematical based techniques that allow the engineer to discharge the possibility of introducing design misinterpretations. Since the resulting formal representation of the system has a unique interpretation, therefore it can be interpreted by a software that allows for automated or semi-automated analysis to discover design flaws, and to validate the result. The introduction of *model checking*, in early 1980s, represented one of the most important achievements in the field of formal methods. In fact, this technique allows for exhaustively and automatically check whether a formal system definition - the model - meets a set of formal requirements. However, while highly promising, model checking required several years to be effectively applied to a real-world scenario and be integrated into a development process.

In the 1990s, the advances of the model-based techniques have received significant interest in the community of safety and reliability engineering. The ensemble of those disciplines is defined as *model-based safety assessment* (MBSA). The objective of this research field is to support the analysis prescribed by the safety assessment process, by relying on the definition of a formal model of the system. In particular, original MBSA techniques [68, 100, 13] were directed to provide a single formalism able to automa-

tize the production of classical safety artifacts such as Fault Trees (FT) and Fault Modes and Effects Analysis (FMEA) tables. However, those approaches were operating only at the safety assessment level, and the relation with the nominal system analysis (i.e., the V-Model) was not considered.

The successive integration with model checking techniques allowed to reduce this gap [29, 39, 38, 42, 11, 22, 21, 37]. However, the resulting techniques were not directed to natively support the distinctive refinement of the design that characterizes the development of a safety critical system. At the same time, they experience significant issues when dealing with real-world, large scale system designs.

Contributions

In this Thesis we define a set of comprehensive model-based safety assessment methodologies and techniques able to overcome the limitations of current approaches. The proposed solution provides i) a seamless integration with standard V-Model and safety assessment processes, ii) able to natively follow the characteristic refinement of the system design, iii) by providing advanced and completely automated techniques for assuring system safety and reliability, iv) while guaranteeing the ability to deal with real-world system designs.

This target has been reached by integrating several different techniques into a single framework. The contributions of this Thesis that support these results are the followings:

- In [30] we improve the performance of the minimal cutsets computation, which represents the basis of all model-based safety assessment techniques that rely on symbolic model checking. This result has been possible via the application of modern SAT-based algorithms. Moreover, we widen the level of expressivity supported by the minimal

cutsets computation, moving from pure invariant definition of system specifications to a full support of Linear Temporal Logic (LTL) [102].

- In [35] we encompass an emerging paradigm called contract-based design (CBD) in order to define a novel methodology that natively supports the refinement of system design. In fact, CBD introduces a formal approach to automatically analyze the correctness of system decompositions into a hierarchy of sub-systems and modules.

- We extend current model-based safety assessment methodologies in order to support the reliability analysis of redundant architectures [33]. This approach integrates Satisfiability Modulo Theory (SMT) and minimal cutsets computation in order to support the analysis in the early stages of the system design e.g., when modules implementation have not yet defined. Moreover, in [34] we apply a specialized technique based on model abstraction that significantly improves the performance.

- We implemented all aforementioned techniques into a set of specialized tools such as nuXmv [97], xSAP [27], and OCRA [54], which are engineered in order to support a comprehensive framework that follows the system design by supporting both V-model and safety assessment processes.

- In order to validate the practical applicability of the methodologies introduced in this Thesis, we applied them to a series of real-world case studies. Most notably, the aforementioned approaches are applied in a joint project with the National Aeronautics and Space Agency (NASA) to formally analyze a series of possible designs for the next generation of the Air Traffic Control system (ATC) [90, 69]. Furthermore, an analysis of the reliability has been applied to the archi-

tectural design of the Primary Flight Computers of the Boeing 777 and Airbus A330. Moreover, we discuss the effectiveness of the proposed approach to produce safety analysis artifacts, by applying it to a case-study described in the Aerospace Information Report [109];

- We provide the whole documentation regarding case studies and tools at the link `www.mattarei.eu/cristian/thesis`.

Structure of the Thesis

The rest of this Thesis is organized as follows:

- Part I provides the background notions that identify the starting point of this Thesis. This Part provides an overview of V-Model and safety assessment processes, their integration with formal methods, and a set of formal definitions characterizing the problem that we intend to solve.
- Part II elaborates on the problem of minimal cutsets computation. The first portion describes how to relate nominal design and its extension with failure behaviors. Previous techniques are then discussed, in addition to a set of simple extensions that can be applied to solve this problem. This Part continues with the introduction of novel techniques that define the new state of the art in the minimal cutsets computation. An extensive experimental evaluation is then described, followed by the description of an LTL extension, and future directions.
- Part III describes the integration of safety analysis with contract-based design. In this Part we follow the description of the technique with a running example taken from an avionic standard. Subsequently, we provide a detailed definition of contract-based design, which is then extended into the contract-based safety analysis approach. This Part

concludes with an experimental evaluation and a discussion on future directions.

- Part IV elaborates on the techniques for the reliability analysis of redundant architectures. Firstly, it provides an overview of the techniques used to increase hardware reliability by the application of components redundancy. Afterwards, we provide the detail of the automated technique based on Satisfiability Modulo Theory, and its subsequent improvement based on predicate abstraction. Experimental evaluations and future directions conclude this Part.

- Part V is devoted at describing the tools architecture that we designed in order to carry out the aforementioned techniques. This Part describes the evolution that have been applied on nuXmv, xSAP, and OCRA tools in order to defined the model-based safety assessment approach described in this Thesis. A discussion on the resulting comprehensive process is then provided.

- Part VI supports the effectiveness of the techniques that we have introduced in this work, by providing the details of their application to a set of real-world case studies. In this Part we first describe the analysis of a triple modular generator, which is a small but representative example to introduce the application of model-based safety assessment. Afterwards, we provide the details of the evaluation of the next generation of the air traffic control system, the analysis of the Fly-by-Wire architectures of two modern aircraft, and an extract of the results reached on the evaluation of an avionic based wheel braking system.

Part I

State of The Practice and Background Notions

Introduction

The development process of a safety critical system is subject to a set of different analyses and methodologies that are combined together into a united framework. Such process, identified as Validation, Verification, and Safety Assessment, is well established in the community of system development. In parallel, the symbolic model checking has emerged in the last decades as a valid technology to prove system correctness, both for hardware and software components.

This Thesis concerns the application of symbolic model checking techniques to the development process of a safety critical system, and in this Part we provide an overview of both disciplines as a background of the Thesis contributions.

This rest of Part I is organized as follows:

- Chapter 2 describes the current development process of a safety critical system;
- Chapter 3 provides an overview of the methodologies that integrate model based analysis in the safety assessment process;
- Chapter 4 defines some technical background relevant for the techniques that we will introduce in this Thesis.

2

Safety Critical Systems Development

The continuous increase in complexity and capabilities of every electrical and mechanical systems is a never ending story. The development of novel and more advanced technologies require more advanced evaluation techniques, able to guarantee the correctness of design and implementation. In fact, the development process of systems like micro processors, motor vehicles, power plants, and aircraft are highly detailed and complex, characterized by multiple levels of sub-processes and analysis. Airbus Industries produces every year an average of 123 millions of pages [113], only for the customers documentation. Essentially, when dealing with such a complex system, only the management of the documentation describing the system and its development requires, per se, a well defined process. In case of aircraft developments, on top of their complexity, the main concern regards safety and reliability aspects.

The development of such a complex safety critical system requires a well defined process that clearly defines each phase of design and implementation, which in turn are divided into several sub-processes and sub-phases. Each of those phases are defined with a clear set of analyses and expected outcomes, whose satisfactory results prescribe the transition to the next phase.

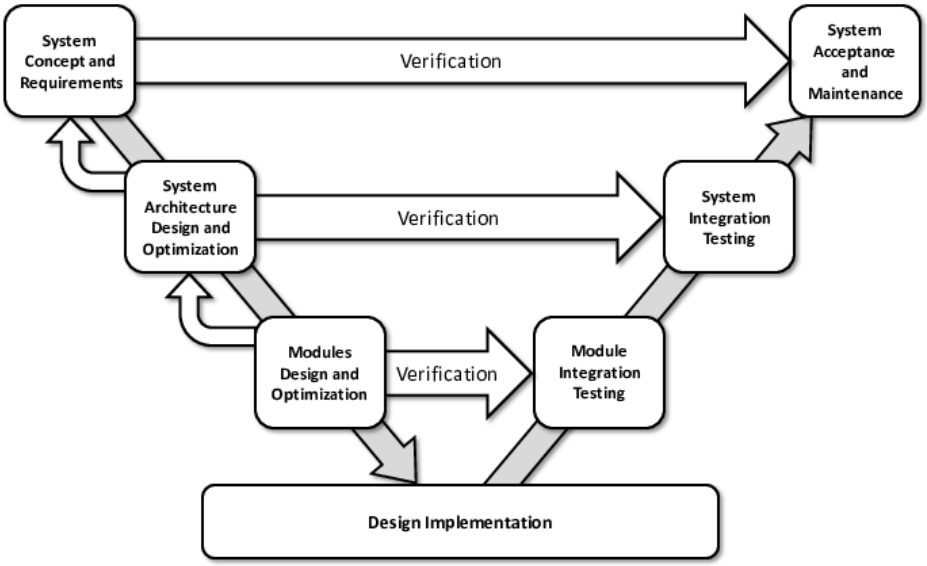


Figure 2.1: Standard V-Model

In general, each field relies on slightly different interpretations of the development process, however in this Thesis we refer to the one provided by the Society of Automotive Engineers (SAE) in the Aerospace Recommended Practice (ARP) [108, 107]. The SAE is an association of engineering professionals devoted to the definition of standard that guide the development of transportation systems.

2.1 The V-Model

The development of a safety critical systems such as aircraft is guided by the V-Model process [108]. The V-Model is a specialization of the waterfall, and it relates each step of the design phase with a post-implementation phase, by defining (in the former) how the requirements are produced, and (in the latter) how they are verified against the implementation. Figure 2.1 shows a graphical representation of the V-Model. The left-side describes the system definition considering requirements, architecture and expected

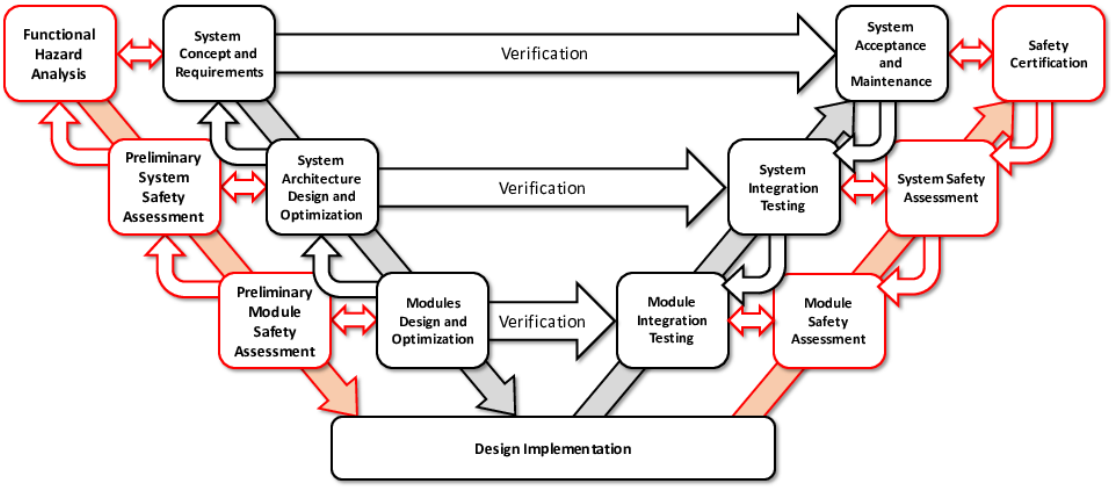


Figure 2.2: Standard V-Model with Safety Assessment

behavior of each single component. Differently, the right-side shows how to evaluate if the system implementation is in accordance with the requirements. The horizontal dimension of the V-Model express which step of system definition (left-side) is in relation to the verification and testing phase (right-side). For instance, the first step "System concept and requirements definition" defines the acceptance criteria for "System acceptance and maintenance".

2.2 Safety Assessment

Safety aspects are paramount in the development of a safety critical systems, thus the V-Model needs to be extended with a specialized process in order to property address this requirement. Such extension is called Safety Assessment.

The Safety Assessment, shown in the red part of Figure 2.2, integrates each phase of the V-Model with a set of specialized activities and analysis aimed at defining the system hazards, their severity and how they are addressed during the design phase. Moreover, the Safety Assessment pre-

scribes also how to verify whether the system implementation meets the safety requirements. More specifically, the Safety Assessment process is defined by the following steps:

- *Functional Hazard Analysis* (FHA): it identifies the potential risks of functional losses by assigning to them a severity degree;
- *Preliminary System Safety Assessment* (PSSA): it derives safety requirements for the subsystems. The PSSA process iterates with the design evolution in order to refine system safety requirements;
- *Preliminary Module Safety Assessment*: it verifies whether the refinements of the sub-systems obey the safety requirements derived in the PSSA phase. This step iterates with the module design evolution in order to refine the sub-systems safety requirements;
- *System and Module Safety Assessment* (SSA): once design implementation is completed, the System Safety Assessment process verifies whether the safety requirements are met in the implemented design.
- *Certification*: the analysis performed in previous steps allow the system to be certified according with the requirements defined early in the process.

Safety Assessment phases are carried out by relying on specific techniques, whose the most relevant ones are Fault Tree Analysis (FTA) and Fault Mode and Effects Analysis (FMEA) [107, 117].

2.2.1 Fault Tree Analysis

Fault Tree Analysis [118, 117] is a deductive technique, whereby an undesired state (the so called *top level event* – TLE) is specified, and the system is analyzed for the possible fault configurations (sets of faults, a.k.a. basic

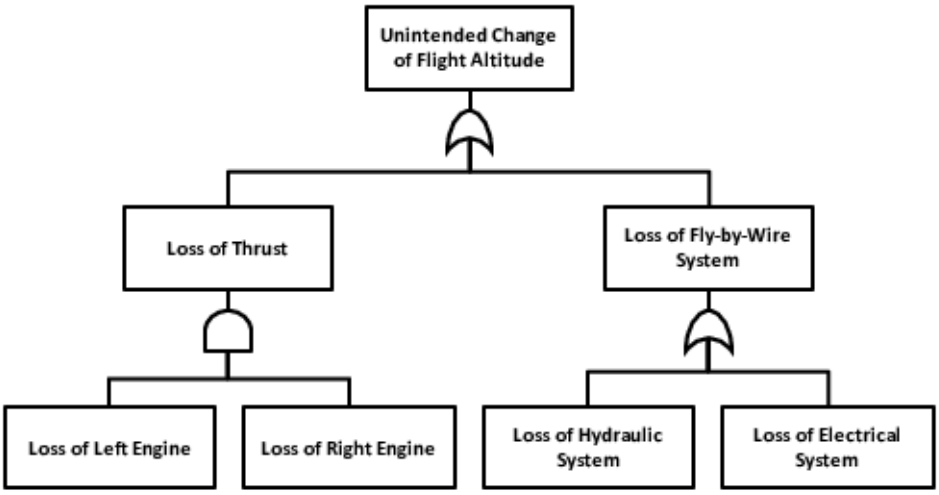


Figure 2.3: Fault Tree Example

events) that may cause the top event to occur. Fault configurations are arranged in a tree, which makes use of logical gates to depict the logical interrelationships. Sub formulas of such tree may represent representative partial failure conditions, due to this fact they are labeled as *intermediate events*. For instance, considering the fault tree in Figure 2.3, the malfunction of left or right aircraft engine represent cause for the intermediate event "Loss of Thrust".

Of particular importance in safety analysis is the list of *minimal* fault configurations, i.e. the *Minimal Cut Sets (MCSs)*. More specifically, a cut set is a set of faults that represents a necessary and sufficient condition that may cause a system to reach an unwanted state/behaviour. For instance, the cut sets in the fault tree of Figure 2.3 are {Loss of Left Engine, Loss of Right Engine}, {Loss of Hydraulic System}, and {Loss of Electrical System}. Moreover, minimality implies that every proper super-set of it cannot prevent the possibility to have the malfunction. Important implication of this aspect is that the Fault Tree collapses to TRUE when the safety hazard is reachable without triggering of any fault. This configuration is

Fault Configuration	Intermediate Events	Top Level Events
"Loss of Hydraulic System"	"Loss of Fly-by-Wire System"	"Unintended Change of Flight Altitude"
"Loss of Electrical System"	"Loss of Fly-by-Wire System"	"Unintended Change of Flight Altitude"
"Loss of Left Engine" and "Loss of Right Engine braking"	"Loss of Thrust"	"Unintended Change of Flight Altitude"

Table 2.1: Example of an FMEA table (w.r.t. FT in Figure 2.3)

represented by the empty set, which is evidently minimal.

2.2.2 Fault Mode and Effect Analysis

FMEA works in a bottom-up fashion, and aims at producing a tabular representation (called *FMEA table*) that represents the causality relationships between (sets of) faults, intermediate events and a list of properties (representing undesired states, as in the case of FTs). Although FMEA is different in spirit from FTA, generation of MCSs can also be used as a building block for computing FMEA tables, in particular under the assumption of monotonicity [41]. For instance, the FMEA table in Figure 2.1 represents the MCS obtained from the FT in Figure 2.3.

2.2.3 Qualitative and Quantitative Analysis

Fault Tree Analysis and FMEA are, in principle, qualitative analyses i.e., the outcome is an unquantifiable measure subjective to a pass/fail result. However, an important aspect of safety assessment is the quantitative evaluation, i.e. the association of basic and intermediate events with probabilities.

In particular, the determination of the probability of the TLE is used to estimate the likelihood of the safety hazard it represents. Such computa-

tion can be carried out by evaluating the probability of the logical formula given by the disjunction of the MCSs. It is standard practice, in particular for complex systems, to consider only cut sets up to a maximum cardinality – in order to simplify the computation. This approach is justified by the fact that, in practical cases, cut sets with high cardinality have low probabilities, and may be "safely" ignored. However, it is essential to have criteria to estimate the error which is inherent in such approximation, since under-approximating the probability of a hazard would not be acceptable.

3

Model-Based Validation, Verification, and Safety Assessment

In the last decades the system complexity has been dramatically increased, and their development and validation consumes an ever-increasing percentage of the total development cost.

This aspect reaches even more importance in the development of a safety critical system, where a hardware or software component failure may lead to a loss of life. Authorities categorize those events as catastrophic, and require them to be so unlikely that their occurrence is not expected during the entire life of the system [9]. In principle, the malfunction of a complex safety critical system can be caused by hardware failures or design flaws. This Chapter concentrates on the Model-Based techniques, which aim at supporting the system design in order to minimize the possibility of introducing design errors. In particular, such techniques support the definition of the system requirements, guarantee that they are met by the high level system design and properly derived into the low level system definition, and correctly implemented by hardware and software components. This approach relies on formal models and specifications in order to guarantee an unambiguous traceable process that links system requirements to system implementation, going through the entire design phases.

Formal Model-Based techniques have become essential, and sometimes even required, in the development of modern complex safety critical systems [83, 95]. The progressive integration of such techniques has been enabled by the significant increase in scalability performance in formal verification [103].

3.1 Formal Validation and Verification

Formal validation and verification is a proof-based methodology to assess the correctness of specifications, system design, and implementation. In principle, formal V&V can be a pure paper-and-pencil activity, thus the use of automated formal analysis guarantees a higher degree of confidence, and reduces the cost and time needed to carry out the proof of correctness. Possible techniques that can be used to perform formal V&V include automated theorem proving, abstract interpretation, and model checking. However, in this Thesis we concentrate on the latter.

3.1.1 Formal Specification

The first step of this process is the formal specification of system requirements. This process is carried out by translating the informal requirements by using a notation derived from mathematical logic.

This transformation guarantees a complete unambiguous interpretation of the system requirements, otherwise not possible when relying on informal languages i.e., English. More specifically, the formal definition of system requirements is composed of: i) the expected behavior and characteristics of the system, ii) the system design itself, and iii) a set of assumption on the environment in which the system has to operate.

For instance, the requirement “A loss of thrust power shall not be caused by a single engine failure” represents a characteristic of an aircraft design,

while “the hydraulic system provides hydraulic power to the Fly-by-Wire system” is prescribing part of the system architecture. Differently, “the system design does not take into account the impact with objects wider than 1 meter” defines an assumption on the environment. Moreover, the formal specification should also consider to provide a reasonable interpretation of the basic environmental assumptions such as the physical laws. This interpretation should be in accordance with the level of abstraction of the formal model e.g., describing that an aircraft cannot be in two different positions at the same time might require only a mutual exclusive condition.

Expected behavior, system design, and environmental assumptions are usually refined during the design process, and they range from a high-level abstract representation of the system, to a detailed implementation of the modules composing it.

3.1.2 Formal Validation

Particularly important in formal analysis is to guarantee that the formal specification fulfills the intended semantics. This check is carried out by the Formal Validation. More specifically, this phase aims at increasing the confidence that the formal specification represents a correct interpretation of the system expected behavior, system design, and environment behavior. In fact, the formal validation analysis should check that the formal specification i) does not contain contradictory or tautological definitions e.g., the expected behavior that “during cruise operation, the aircraft altitude shall be always greater than 35000 feet” is in contradiction with “the pilot can always be able to perform the intended maneuver”; and ii) that it provides a correct interpretation of the definitions that are intended to be refined. In fact, in a refinement based process - as in the V-Model - the first step formalizes the conceptual requirements, while the successive ones iteratively refine the formalization of the preceding high level requirements.

All analysis described in this phase are usually performed over both whole formal specification, and on some relevant sub parts. For instance, considering that we want to analyze the ability of an Air Traffic Control system to avoid mid-air collisions between two aircraft. In this case, the formal validation phase should guarantee that the environment conditions allow two aircraft to be at the same position at the same time.

3.1.3 Formal Verification

Formal verification is the use of proof methods to prove that, given the environmental assumptions, the formal design of the system fulfills the requirements. The artifacts produced by this analysis are subjected to an evaluation, ensuring that they match the expected result. In fact, the positive result prescribes the possibility to move to the next refinement phase. In case of an unsatisfactory result, the process prescribes to reconsider the interpretation expressed by the formal specification, and revise it accordingly.

3.2 Model-Based Safety Assessment

Model-based safety assessment is a natural extension to the formal V&V by integrating safety aspects of the overall process into the formal analysis framework. This approach, which is less consolidated than the formal V&V, aims at supporting the production of safety artifacts, like Fault Trees and FMEA tables, by relying on formal techniques.

The model-based safety assessment mimics the steps that characterize the V&V process. In particular, it prescribes to i) define a set of formal safety requirements, ii) formalize the system failure conditions, and iii) evaluate whether the system satisfies the safety requirements.

Considering that the outcomes of the safety analysis are (usually) Fault

Trees and FMEA tables, therefore the safety requirements express expected properties over those artifacts. For instance, conditions like "single failures cannot cause a Loss of Thrust" and "the probability of an Unintended Change of Flight Altitude should be less than 10^{-9} per flight hour" are possible safety requirements.

In standard model-based safety assessment, the formalization of system failure conditions usually rely on a single failure model that describes how each component may fail, and which might be the resulting implications. Afterwards, all failure conditions are combined together and represented in the form of Fault Trees and FMEA tables.

Afterwards, the analysis of whether the system satisfies the safety requirements are directly derived from an inspection of the safety artifacts produced in the previous phase. For example, the Fault Tree represented in Figure 2.3 satisfies the safety requirement "single failures cannot cause a Loss of Thrust", in fact that event requires "Loss of Left Engine" and "Loss of Right Engine" to occur.

More advanced model-based safety assessment techniques integrate more tightly the formal V&V process. In this case, the effects that derive from a component failure are directly extracted from the formal specification defined in the V&V phases. This approach has an evident advantage of relying on a single model for both nominal and safety analysis, thus minimizing the possibility of introducing errors in the formalization phase.

4

Technical Background

This Chapter provides the technical background representing the starting point of all techniques introduced in this Thesis. In particular, Sections 4.1 and 4.2 elaborate on the notions of Satisfiability Modulo Theory (SMT) and Binary Decision Diagrams (BDD), which represent the bases for the formal representation of a system. Section 4.3 defines the problem of Symbolic Model Checking that will be applied both to finite and infinite state systems, representing also the main technology used in this Thesis to instantiate a formal validation, verification, and safety assessment process.

4.1 Satisfiability Modulo Theory

The Boolean Satisfiability Problem (SAT) is the problem of determining if there exists a total truth assignment, to a given propositional Boolean formula, that evaluates to TRUE. In general terms, a propositional formula is an arbitrary combination of conjunction or disjunction of literals (e.g., A and $\neg A$ are literals). However, every propositional formulas can be reduced into Conjuntive Normal Form (CNF), which consists in a conjunction of disjunction of literals, where each disjunctive is called clause. Differently, a Disjunctive Normal Form (DNF) represents a propositional formula as a disjunction of conjunction of literals. For instance, $A \vee (B \wedge C)$ is in DNF,

while $(A \vee B) \wedge (A \vee C)$ is the CNF counterpart.

The Satisfiability Modulo Theory (SMT) is an extension of the SAT decision problem, where the formula is not purely Boolean, but it is expressed in a combination of theories expressed in first order logic with equality. The definition of an SMT problem, as in SAT, is a conjunction of clauses, but in this case each literal can be expressed as a predicate over non Boolean variables. Examples of common used theory in verification are the following:

- Equality and Uninterpreted Functions ($\mathcal{EUF}$):

  $((x = y) \wedge (y = f(z))) \rightarrow (g(x) = g(f(z)))$;

- Difference logic ($\mathcal{DL}$):

  $((x = y) \wedge (y - z \leq 4)) \rightarrow (x - z \leq 6)$;

- Linear arithmetic over the rationals ($\mathcal{LA}(\mathbb{Q})$):

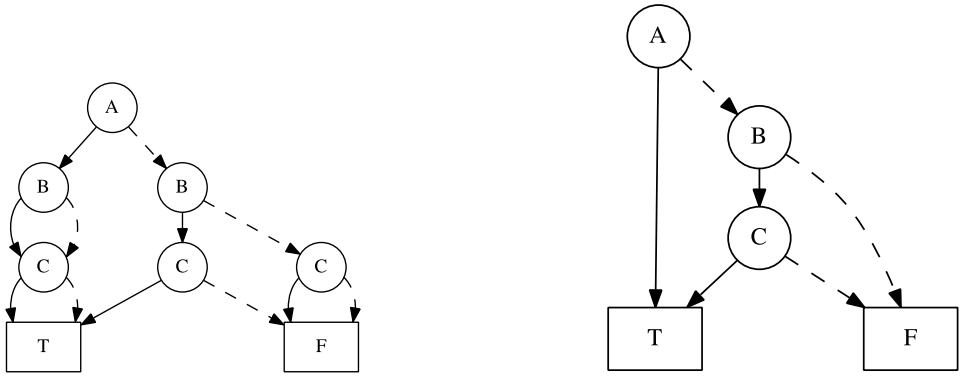
  $(T_\delta \rightarrow (s_1 = s_0 + 3.4 * t - 3.4 * t_0)) \wedge (\neg T_\delta \rightarrow (s_1 = s_0))$;

- Linear arithmetic over the integers ($\mathcal{LA}(\mathbb{Z})$):

  $(x := x_l + 2^{16} x_h) \wedge (x \geq 0) \wedge (x \leq 2^{16} - 1)$.

While a propositional Boolean formula can be satisfied by a finite number of possible models, in case of an SMT formula this set may be infinite. For instance, the SMT formula $(x > 0)$, with x being a Rational variable, is satisfiable for every value of x greater than 0.

Analogously to SAT/SMT, the All-SAT/All-SMT problem consists in determining all satisfiable assignments of a given propositional/SMT formula.



(a) Complete (b) Reduced

Figure 4.1: BDD example of the formula $A \vee (B \wedge C)$

4.2 Ordered Binary Decision Diagrams

An (Ordered) Binary Decision Diagram, namely BDD, is a data structure used to represent a Boolean formula. In particular, a BDD, as shown by the example in Figure 4.1, is a single rooted binary acyclic graph, with internal decision nodes and two final nodes (i.e., TRUE and FALSE). Each decision node N has two outgoing edges, *then* and *else* edge. Those edges differ in the semantics, in fact taking the *then* edge means to assign the variable that labels N to TRUE, while FALSE in case of the *else* one. Intuitively, a path from the root to the TRUE node represent a valid assignment to the formula, while the paths leading to FALSE are not valid assignments. Moreover, the order of which the variables are visited in a path is called *variable ordering*.

Each path in the BDD in Figure 4.1a visits every label. However, nodes like the right most C are not necessary to represent a satisfiable assignment of the formula. In fact, both outgoing edges end up to the same node, thus they can be safely "removed". Applying this operation until all nodes have separate destinations for their outgoing edges generates reduced BDD, as represented in Figure 4.1a. A very important characteristic of this structure

is that, given a variable ordering, two formulas produces the same BDD if and only if they are equivalent. Thus, a BDD is a canonical form of the models satisfying a formula.

4.3 Symbolic Model Checking

This section describes a formal framework that is commonly used to describe transition based systems, and formal requirements. Those formal concepts are then linked together by the introduction of the Symbolic Model Checking, which instantiate the problem of verifying whether a transition system obeys or not a formal requirement.

4.3.1 Symbolic Transition System

In this work we concentrate on systems that can be represented as *Symbolic Transition Systems* (STS). This formalism, as in Definition 4.3.1, describes the behavior of a discrete system characterized by a set of possible initial states, and a transition relation that expresses how the system evolves from a state to the next one. In the context of a transition relation, the states before and after the transition are usually called respectively *current* and *next*.

In this case, the transition system is called *symbolic* because the set of states are represented with a formula (for the shake of clarity, we consider it as Boolean). This approach allows for a concise representation of the system states, in fact they are defined as the satisfiable assignments of that formula.

Definition 4.3.1 (Symbolic Transition System)**.** A *Symbolic Transition System* is a tuple $S = \langle V, I, T \rangle$ where V is a set of (state) variables, $I(V)$ is a formula representing the initial states, and $T(V, V')$ is a formula rep-

resenting the transitions. A *state* of S is an assignment to the variables V.

A symbolic transition system describes the evolution of a system as an initial state and a series of transitions that link a state to the next one. Such, possibly infinite, executions are called *traces* (see Definition 4.3.2).

Definition 4.3.2 (Trace)**.** A *trace* of a symbolic transition system $S = \langle V, I, T \rangle$ is a finite or infinite sequence $\pi = s_0, s_1, s_2, \ldots$ of states such that $s_0 \models I$ and $\forall_{i \geq 0} s_i, s_{i+1} \models T$. Moreover, the state s_i with $i \geq 0$ of a trace $\pi = s_0, s_1, s_2, \ldots$ is denoted as $\pi[i]$, and the assignment to the variable $x \in V$ at the state s_i is expressed as $\pi[i](x)$.

An important concept for a symbolic transition system is the *reachable states*. Formally defined in 4.3.3, a reachable state is a state that can be reached by keep applying the transition relation starting from the initial states.

Definition 4.3.3 (Reachable States)**.** Given a *Symbolic Transition System* $S = \langle V, I, T \rangle$, a state s is reachable in S if and only if there exists a trace $\pi = s_0, s_1, \ldots, s$ of S. The set of reachable states of S is defined as $Reach(S) = \{s \models V \mid s \text{ is a reachable state}\}$. The set $Reach(S)$ can be symbolically represented with the formula $\bigvee_{s \in Reach(S)} s$. Moreover, if the set $Reach(S)$ is finite/infinite then S is a *finite/infinite states transition system*.

A state without any successor (i.e., next state) is called a deadlock state. While the semantics of the symbolic transition systems allows for the definition of a system with deadlocks, in practice this is not desirable since it moves from a system with infinite to finite traces length. This aspect has an implication on the techniques that can be used to analyze the system, however those details are not covered here.

Definition 4.3.4 (Deadlock-free Symbolic Transition System)**.** A Symbolic Transition System $S = \langle V, I, T \rangle$ is called *deadlock-free* when for all traces $s_0, s_1, \ldots, s_k$ of S there exists a state s_{k+1} of S such that $s_k, s_{k+1} \models T$.

Given a state, namely the current, and applying the transition relation, we then obtain all states that have distance 1 from the original. Intuitively, this concept can be extended in general terms to n unrollings, as in Definition 4.3.5, to introduce the concept of state distance.

Definition 4.3.5 (States distance)**.** Given a *Symbolic Transition System* $S = \langle V, I, T \rangle$, the distance between two reachable states s^1 and s^2 of S, namely $Dist(S, s^1, s^2)$, is the length of the shortest trace π in S such that $\pi = s^1, \ldots, s^2$.

The states distance allows us to introduce the *diameter* of a symbolic transition system as in Definition 4.3.6. This concept, particularly important when analyzing the behavior of a transition system, allows for distinguish between different classes of transition systems. Intuitively, a system with a deep diameter requires a significant amount of transition relation unrollings in order to analyze all possible behaviors. Differently, a small diameter may implies less effort during the system analysis.

Definition 4.3.6 (System Diameter)**.** Given a *Symbolic Transition System* $S = \langle V, I, T \rangle$, the diameter of S is k if $\exists_{s,s' \in Reach(S)} Dist(S, s, s') = k$ and $\forall_{s,s' \in Reach(S)} Dist(S, s, s') \leq k$

An important class of systems are the *combinatorial* ones, as defined in 4.3.7. Particularly used in circuits and architecture definition, a combinatorial system represents a memoryless evolution, whose next states are not dependent to the current ones. A combinatorial system can be represented in the framework of a symbolic transition systems as a system with diameter equal to 0, meaning that the reachable states of the system are (only) defined by the constraints on the initial ones.

Definition 4.3.7 (Combinatorial System)**.** A Symbolic Transition System $S = \langle V, I, T \rangle$ is called *combinatorial* when the diameter of S is 0.

The abstract representation and formalization of a real system, via symbolic transition system, may require to get composed with other systems. The composition of two systems can be performed with different approaches, however in this work we concentrate on synchronous ones. Definition 4.3.8 introduces the notion of *Synchronous Product of Symbolic Transition Systems*, where the resulting system is the conjunction of the formulas representing both initial states and transition relation. This semantics imposes that each transition has to satisfy both systems interpretation. Other approaches on systems compositions, like the asynchronous ones, are not considered in this work.

Definition 4.3.8 (Synchronous Product of Symbolic Transition Systems)**.** Given two Symbolic Transition Systems $S' = \langle V', I', T' \rangle$ and $S'' = \langle V'', I'', T'' \rangle$, the *synchronous product* $S = \langle V, I, T \rangle$ of S' and S'' is a Symbolic Transition System where $V = V' \cup V''$, $I = I' \wedge I''$, and $T = T' \wedge T''$.

4.3.2 Linear Temporal Logic

The formal design process of a system is characterized by the definition of the system itself and a set of expected behaviors e.g., system requirements. The framework described in this work relies on symbolic transition system for the system definition, and Linear Temporal Logic for the definition of the expected behavior.

The *Linear Temporal Logic* (LTL) is a modal temporal logic that can be used to encode properties over system traces. The syntax of LTL, defined in 4.3.9, is quite simple and includes four operators (i.e., **X**, **F**, **G**, and **U**) in addition to the pure propositional ones (i.e., $\wedge$, $\vee$, and $\neg$).

Definition 4.3.9 (Syntax of Linear Temporal Logic)**.** LTL formulae over the set V of variables are formed according with the following grammar:

$$\varphi ::= a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \varphi_1 \mathbf{U} \varphi_2$$

where $a \in V$.

The semantics of the LTL is defined in the context of system traces (see Definition 4.3.10). A part from the propositional operators, the temporal ones can intuitively described as follows:

- $\mathbf{X}\varphi$ or *next*: φ holds in the next state;
- $\mathbf{F}\varphi$ or *finally*: φ holds at some point in the future;
- $\mathbf{G}\varphi$ or *globally*: φ holds at every points in the future;
- $\varphi_1 \mathbf{U} \varphi_2$ or *until*: φ_1 holds for every states until φ_2 holds.

Definition 4.3.10 (Semantics of Linear Temporal Logic)**.** Given an LTL formula φ and a trace $\pi = s_0, s_1, s_2, \ldots$, we define that the formula φ is true at an instant $i \geq 0$ of π, denoted as $\pi, i \models \varphi$, as follows:

- $\pi, i \models a$ iff $\pi[i] \models a$
- $\pi, i \models \neg\varphi$ iff $\pi, i \not\models \varphi$
- $\pi, i \models \varphi_1 \wedge \varphi_2$ iff $\pi, i \models \varphi_1$ and $\pi, i \models \varphi_2$
- $\pi, i \models \varphi_1 \vee \varphi_2$ iff $\pi, i \models \varphi_1$ or $\pi, i \models \varphi_2$
- $\pi, i \models \mathbf{X}\varphi$ iff $\pi, i+1 \models \varphi$
- $\pi, i \models \mathbf{F}\varphi$ iff $\exists j \geq i$ such that $\pi, j \models \varphi$
- $\pi, i \models \mathbf{G}\varphi$ iff $\forall j \geq i$ such that $\pi, j \models \varphi$

- $\pi, i \models \varphi_1 \mathbf{U} \varphi_2$ iff $\exists j \geq i$ such that $\pi, j \models \varphi_2$ and $\forall_{0 \leq k < j}\ \pi, k \models \varphi_1$

The satisfaction relation $\models$ between π and φ, in notation $\pi \models \varphi$ (namely, π models φ), holds if $\pi, 0 \models \varphi$ holds.

Some patterns of LTL properties are of particular interest. In fact, they are known to be commonly used when modeling a reactive system. The main patterns in LTL properties definition can be categorized in one of the following sets:

- *Safety*: "a bad condition never happens", $\mathbf{G}(\neg bad)$;
- *Liveness*: "at some point something good happens", $\mathbf{F}(good)$;
- *Fairness*: "φ holds infinitely many times", $\mathbf{G}(\mathbf{F}(\varphi))$;
- *Invariant*: "φ holds in every state of the system", $\mathbf{G}(\varphi)$.

4.3.3 Model Checking

The introduction of the formal representation of a system (i.e., STS), and a property specification language (i.e., LTL properties) calls for an approach to evaluate whether the first meets the latter. This problem, formally defined in 4.4.2, is called *LTL model checking*. More specifically, we want to check whether all traces of an STS model a given LTL property.

Definition 4.3.11 (LTL Model Checking Problem)**.** Given a *Symbolic Transition System* $S = \langle V, I, T \rangle$, and an LTL formula φ, the *LTL model checking problem* denoted with $S \models \varphi$ is the problem of checking if for all traces $\pi = s_0, s_1, \ldots$ of S, $\pi \models \varphi$.

An important subset of LTL model checking is the invariant model checking. This problem consists into evaluating if a given propositional property holds in every reachable states of a symbolic transition system.

As defined in 4.3.12, the invariant model checking problem can be defined as an instance of LTL model checking.

Definition 4.3.12 (Invariant Model Checking Problem)**.** Given a *Symbolic Transition System* $S = \langle V, I, T \rangle$, and a propositional logic formula φ, the *invariant model checking problem* is the LTL model checking problem of $S \models \mathbf{G}\varphi$.

The *reachability problem* is the problem of checking whether a particular condition can be reached by a Symbolic Transition System. Defined in 4.3.13, this problem can be encoded into LTL model checking.

Definition 4.3.13 (Reachability Problem)**.** Given a *Symbolic Transition System* $S = \langle V, I, T \rangle$, and a propositional logic formula φ, the *reachability problem* is the LTL model checking problem of checking $S \models \mathbf{F}\varphi$.

4.3.4 LTL Satisfiability

The definition of a set of formal requirements sometimes need specific evaluations in order to assure that they are consistent i.e., not contradictory. The *LTL Satisfiability Problem* (Definition 4.3.14) formalizes this concept, which consists in evaluating whether a single trace can satisfy a given LTL formula. In practice, this problem can be reduced to model checking by introducing the concept of the *universal model* i.e., a model that accepts any possible trace. More specifically, given an LTL formula φ, $SAT(\varphi)$ holds if and only if $\mathcal{U} \models \varphi$, where $\mathcal{U}$ is the universal model.

Definition 4.3.14 (LTL Satisfiability Problem)**.** Given an LTL formula φ, and an LTL formula φ, the *LTL satisfiability problem* denoted with $SAT(\varphi)$ is the problem of checking if there exists a trace $\pi = s_0, s_1, \ldots$ such that $\pi \models \varphi$.

4.4 Symbolic Parameter Synthesis

In some cases, the formal analysis of a system requires more detailed result than the one provided by standard symbolic model checking. In particular, a system S might not model a property φ (i.e., $S \not\models \varphi$), but one would be interested into understanding under which conditions this does not hold. This is the parametric model checking problem, formally defined in 4.4.2, it consists of finding the assignments to a given set of parameters (subsets of the system variables) such that the system models the given property. The satisfiable assignments are expressed as a formula over the parameters (also called, the parameters region). More specifically, a configuration of the parameters belongs to the parameters region if and only if the system models the formula φ or, at some point, it violates the configuration assignment of the parameters.

Definition 4.4.1 (LTL Parameter Synthesis Problem)**.** Given a *Symbolic Transition System* $S = \langle V, I, T \rangle$, a set of Boolean parameters $P \subseteq V$, and an LTL formula φ, the *LTL parameter synthesis problem* denoted with $R(P) : S \models \varphi$ is the problem of finding a propositional formula $R(P)$, called region, such that for every assignment ρ of P it holds that $\rho \models R(P)$ iff $S \models \mathbf{G}(\rho) \rightarrow \varphi$.

Similarly as in standard Model Checking, a sub class of the LTL parameter synthesis problem is its counter part that considers only propositional properties i.e., invariants. This problem is formalized in Definition 4.4.2.

Definition 4.4.2 (Invariant Parameter Synthesis Problem)**.** Given a *Symbolic Transition System* $S = \langle V, I, T \rangle$, a set of parameters $P \subseteq V$, and a propositional formula φ, the *invariant parameter synthesis problem* is the LTL parameter synthesis problem $R(P) : S \models \mathbf{G}(\varphi)$.

4.5 Minimal Cutsets Computation

This Thesis concentrates on the analysis of systems under failure conditions i.e., when some components do not behave correctly. This analysis, formally defined in 4.5.1, build upon parameter synthesis, namely the *Cutsets Computation*. In this case, the parameters are purely Boolean and represent the occurrence of a system failure. Such parameters are called *fault variables*, and linked to a system failure, their semantics expresses that a fault variable becomes TRUE when a failure has occurred. The result of a cutset computation is a set of faults configurations whose activation allows the system to satisfy an unwanted event called *Top Level Event*. In a parameter synthesis setting, this can be reduced in the computation of the negated parameter region that allows the system to not model the Top Level Event.

In this Thesis we consider also the cutsets computation over an LTL formula ϕ (instead of a system S) i.e., $CS(\phi, \varphi, F)$. As for the LTL satisfiability, this problem can be reduced to the cutsets computation over a symbolic transition system as $CS(\mathcal{U}, \phi \wedge \varphi, F)$, where $\mathcal{U}$ is the universal model.

Definition 4.5.1 (LTL Cutsets Computation over STS)**.** Given a *Symbolic Transition System* $S = \langle V, I, T \rangle$, a set of Boolean faults $F \subseteq V$, and an LTL formula φ, whose negation is called Top Level Event, the *LTL Cutsets Computation* is the problem of computing the set

$$CS(S, \varphi, F) := \{FC \in 2^F \mid \exists R(F) \text{ s.t. } R(F) : S \models \varphi \text{ and } (\bigwedge_{f \in FC} (f = \top) \bigwedge_{f' \in F \setminus FC} (f' = \bot)) \models \neg R(F)\}.$$

Definition 4.5.2 extends the concept of cutsets computation to invariant properties. Even if this problem has been defined here as parametric model

checking over LTL formulas, the original and most common approaches consider only the invariant case [104, 63, 37].

Definition 4.5.2 (Invariant Cutsets Computation over STS)**.** Given a *Symbolic Transition System* $S = \langle V, I, T \rangle$, a set of Boolean faults $F \subseteq V$, and a propositional formula φ, which negation is called Top Level Event, the *Invariant Cutsets Computation* is the problem of computing the set $CS(S, \mathbf{G}\varphi, F)$.

In the analysis of system under failure, the fault variables are commonly assumed to follow a monotonic behavior. Formally defined in 4.5.3, this condition applies when for every faults configuration in the cutsets also its supersets belong to the cutsets.

Definition 4.5.3 (Monotonic Faults)**.** Given a *Symbolic Transition System* $S = \langle V, I, T \rangle$, a set of Boolean faults $F \subseteq V$, and a propositional formula φ, called Top Level Event, F in S are called *Monotonic* iff

$$\forall FC \in CS(S, \varphi, F), \{FC' \in 2^F \mid FC \subset FC'\} \subset CS(S, \varphi, F).$$

Considering only the cutsets that are minimal allows us to reduce the size of the faults configurations that have to be considered. This concept, formalized in Definition 4.5.4, is widely used in safety analysis, and it permits to generate a smallest set of cutsets while preserving the same level of expressiveness. Moreover, the minimal cutsets computation is usually applied also even if the fault variables do not follow a monotonic behavior. In fact, the set of MCS represents an over approximation of the CS (i.e., $CS \subseteq MCS$), and in practice this means that the MCS may express a "pessimistic" result.

Definition 4.5.4 (Minimal Cutsets Computation)**.** Given a *Symbolic Transition System* $S = \langle V, I, T \rangle$, a set of Boolean faults $F \subseteq V$, and a propositional formula φ, called Top Level Event, the *Minimal Cutsets Computation*

is the problem of computing the set

$$MCS(S, \varphi, F) = \{cs \in CS(S, \varphi, F) \mid \nexists cs' \in CS(S, \varphi, F), cs' \subset cs\}.$$

The set of Minimal Cutsets is sometimes represented as a formula in Disjunctive Normal Form. With reference to Definition 4.5.4, the resulting formula is as follows:

$$MCS^{\top}(S, \varphi, F) = \bigvee_{FC \in MCS(S,\varphi,F)} (\bigwedge_{fc \in FC} fc)$$

4.6 Fault Trees Representation

A *Fault Tree*, as described in Section 2.2, is a commonly used syntax to represent the combination of system faults that can cause the occurrence of a Top Level Event property. Its standard reprsentation is similar to a propositional formula over system failures, but without the negation operator. The Fault Tree definition allows also for labeling sub parts of the tree, called intermediate events, which is used to describe partial failure conditions. Definition 4.6.1 introduces the syntax of a Fault Tree.

Definition 4.6.1 (Fault Trees representation)**.** A hierarchically organized *Fault Tree* over a set of basic Boolean fault variables BE, and intermediate events IE is formed according with the following grammar:

$$FT ::= be \mid ie \mapsto FT \mid FT \wedge FT \mid FT \vee FT$$

where:

- $ie \in IE$, is a name
- $be \in BE$, is a Boolean fault variable

An important aspect that emerges from Definition 4.6.1, is that a Fault Tree can be reduced to a propositional formula. This transformation is

possible by substituting all occurrence of $ie \mapsto FT$ into FT, and simply discharging the intermediate events labeling. Moreover, a computed set of minimal cutsets can be trivially represented as a Fault Tree (see Definition 4.6.2), and vice versa (see Definition 4.6.3).

Definition 4.6.2 (Minimal Cutsets as Fault Tree)**.** Given a set of minimal cutsets MCS, their Fault Tree representation is:

$$\bigvee_{FC \in MCS} \left(\bigwedge_{fc \in FC} fc \right)$$

Definition 4.6.3 (Fault Tree as Minimal Cutsets)**.** Given a Fault Tree FT the resulting set of minimal cutsets is generated by the following function:

$$\mu(FT) = \begin{cases} \{\{be\}\} & \text{if } FT = be \\ \mu(FT') & \text{if } FT = ie \mapsto FT' \\ \mu(FT') \times \mu(FT'') & \text{if } FT = FT' \wedge FT'' \\ \mu(FT') \cup \mu(FT'') & \text{if } FT = FT' \vee FT'' \end{cases}$$

4.7 Symbolic Model Checking Techniques

In 1990, the techniques described in [48, 92] introduced the first version of a BDD-based symbolic model checker for temporal logic. This approach represented the enabling result for a practical application of model checking techniques. However, BDDs, and more in general the BDD-based model checking, suffer from an explosion in memory consumption when the model reaches a considerable size (e.g., in the order of hundreds of variables). Tools implementing BDD-based CTL and LTL model checking are NuSMV [52], ABC [45], and SAL [18]

The successive introduction of the *Bounded Model Checking* (BMC) [24] represented a possible solution to the issues of BDD-based approaches. In fact, this approach is much less aggressive, and almost linear, from

the memory consumption point of view. More specifically, the BMC is a propositional logic encoding that permits to perform the bounded formal verification using a SAT solver. The BMC technique provides the possibility to check whether a property described in LTL logic holds in a model until a specific bound k. The bound of the verification refers to the evolution steps of the State Machine. This fact imposes that the Bounded Model Checking can answer to the problem with: i) the property does not hold, proved by a counterexample trace witnessing the violation; ii) the property has not been falsified in k steps. The latter result is given when the model checker is not able to prove that, for the paths of length bigger than k, the property holds or not.

Recent extensions of BMC allow also for proving the satisfiability of a subset of LTL, namely the invariant properties. The first relevant work is based on proving safety properties using k-induction [111] i.e., with base and induction (with k steps) proofs. The key idea of k-induction consists in relying on the loop free encoding of the transition relation unrolling. In particular, this approach avoids visiting new states i.e., equal to the ones already visited. More specifically, it uses a SAT solver to prove the conditions that, after k unrollings of a *loop free* path, i) the property cannot be violated from the initial states, or ii) the property cannot be violated from any state. When analyzing a finite state system, this technique guarantees termination.

Successive evolution on SAT-based model checking has been proposed in [93] with interpolation based techniques. This concept encompasses the Craig's interpolants, and they are used to represent an over approximation of the reachable states. This approach works better, in practice, than the induction based, due to the fact that it allows for a shortened unfolding of the transition relation.

More recently, in [43] a SAT-based model checking for safety properties

has been introduced. This method, called Property Directed Reachability (PDR) or IC3 (Incremental Construction of Inductive Clauses for Indubitable Correctness), does not require unrolling of the transition relation. In fact, IC3 operates over an ordered list of k frames that represent an over approximations of the states reachable in k steps or less. The algorithm can terminate by either providing an *inductive invariant* as a proof of satisfiability, or a counterexample to the model checking problem. In this context, and given a model checking problem $S \models \varphi$, an *inductive invariant* is a propositional formula ϕ such that i) $I \models \phi$, ii) $\phi \wedge T \models \phi'$, and iii) $\phi \models \varphi$.

SAT-based LTL and safety properties model checking is implemented by NuSMV2 [51], nuXmv [50], ABC [45], and SAL [18].

Part II

Symbolic Techniques for Minimal Cutsets Computation

Introduction

The minimal cutsets computation is the routine on which all model-based safety assessment techniques, that rely on symbolic model checking, build upon.

As described in Chapter 2, the model-based safety assessment approach is based on the definition of a single model representing the system, which in this case it is described as a symbolic transition system. Applied in different flavors, the symbolic minimal cutsets computation represents the most strategic technique whose improvement allows us to push the performance of symbolic model-based safety and reliability analysis.

The concept of cutsets, which is strongly related to fault tree analysis, has been used for decades by safety engineers on safety critical system development. However, the growing interests in automated techniques able to support those analysis exhibited the need for a generally accepted formalization. The usage of Binary Decision Diagrams to manipulate fault trees and efficiently extract minimal cutsets [62, 104] have led to formalize such operation in terms of prime implicants [105] i.e., the most general (with less literals) implicants of a formula. Those results, combined with the increasing efficiency of symbolic model checking techniques, allowed for the definition of specialized approaches for BDD-based symbolic minimal cutsets computation on dynamical systems [37, 40].

The introduction of SAT-based model checking [24] and successive interpolation based extensions [93] opened up the ability to deal with complex

problems. Most recently, IC3/PDR [44, 43] technique has demonstrated to be the most effective to approach industrial problems.

The need for more efficient and effective techniques expressed in the symbolic model checking field has guided the evolution from BDD-based verification to IC3/PDR. The same need can be found in symbolic model-based safety analysis. In this Part we aim to fill this gap, from the BDD-based approaches proposed in [37] to the most advanced anytime techniques, representing our contribution in this field [30], that build upon SAT and IC3/PDR based model checking.

The rest of Part II is organized as follows:

- **Background**
  - Chapter 5 elaborates on the techniques used to extend the formal model in order to describe system failures;
  - Chapter 6 describes the BDD-based techniques that represent the starting point of the work presented here;
- **Contributions**
  - Chapter 7 defines further improvements as adaptations of existing approaches;
  - Chapter 8 introduces the contribution on efficient techniques that merges scalability efficiency and anytime minimal cutsets generation;
  - Chapter 9 shows a performance comparison between the techniques described in this Part;
  - Chapter 10 introduces an extension to manage Top Level Events defined in Linear Temporal Logic;

– Chapter 11 concludes with an overview of further extensions.

5

Formal Model Extension Techniques

The primary aim of the formal validation and verification is to aid the system development to reduce design flaws, and increase the confidence that the system follows the intended behavior. The formal safety assessment elaborates on this philosophy by analyzing the system behavior under failure conditions. In fact, the formal safety assessment is performed downstream of the standard formal V&V process.

The formal V&V process, as well as the formal safety assessment, aims to avoid error-prone approaches by guaranteeing an internal soundness and correctness i.e., the artifacts are produced in a deterministic way and they are determined only by model and properties. However, the problem that occurs when dealing with those techniques is to guarantee the trustworthiness of the input i.e., that model and properties are an effective abstraction of real system representation and requirements. Similar criticalities characterize also the formal safety assessment, however those issues can be mitigated with a tight integration between the two processes.

The link between formal V&V and SA is represented by the *model extension*. This phase consists of enriching the system components with a behavior that deviates from the nominal one, which represents how that component will operate under failure conditions. The selection between

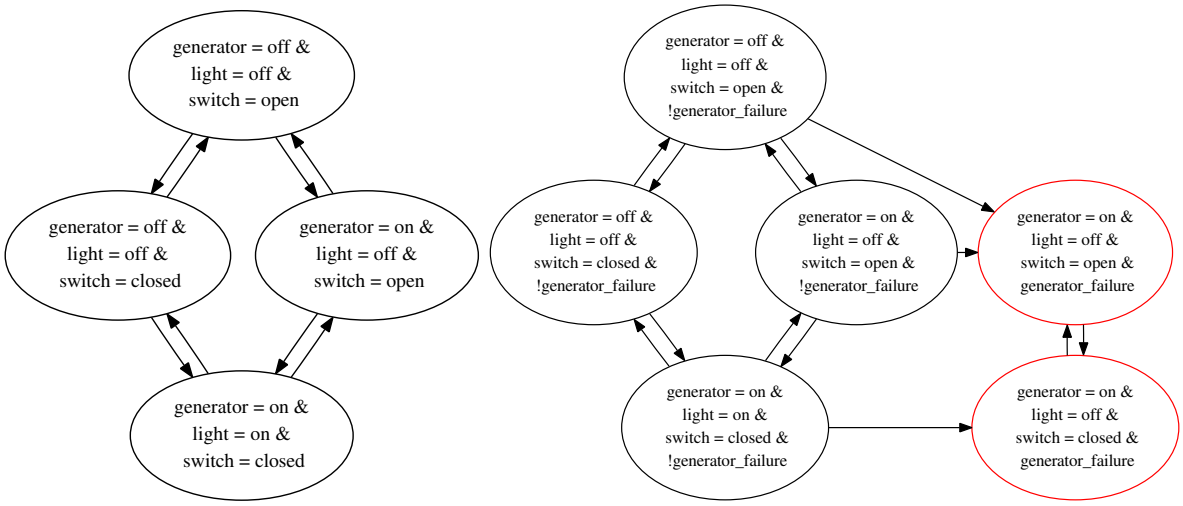
nominal and faulty behaviors is triggered by a particular variable called fault event that can be arbitrarily activated or disabled. In the following sections we firstly provide a formal characterization of the model extension, and afterwards we discuss the main techniques implementing this methodology.

5.1 Formal Characterization

As defined in Chapter 4, in this work we model the system as a Symbolic Transition System (STS). This formalism applies also when describing a system under failure conditions. In this case a subset of the state variables are meant to describe whether the state is a *failure* one or not.

The usual flow of the safety assessment consists in defining a nominal model and afterwards extending it with faulty behaviors. There are different techniques that can be applied in order to extend a model, nevertheless all of them have to produce a result that is compliant with the definition of model extension 5.1.1 . More specifically, a model extension enriches a transition system by adding new (faulty) states and transitions from and to those states. The important property of an extended model is to guarantee that, in absence of faults, its behavior is equivalent to the nominal one. Formally, for every traces of the nominal model there exists an equivalent trace in the extended one where the fault variables are set to false.

Figure 5.1a shows an example of the system evolution of a nominal model. In this simple example a light is turned on when generator and switch are respectively turned on and closed. However, it might be possible that the generator breaks, thus it does not provide energy even if it is turned on. This behavior is shown in Figure 5.1b. In this case, the extended model preserves the nominal behavior, in fact, this transition system is equivalent to the nominal one when *generator_failure* is assigned to false.



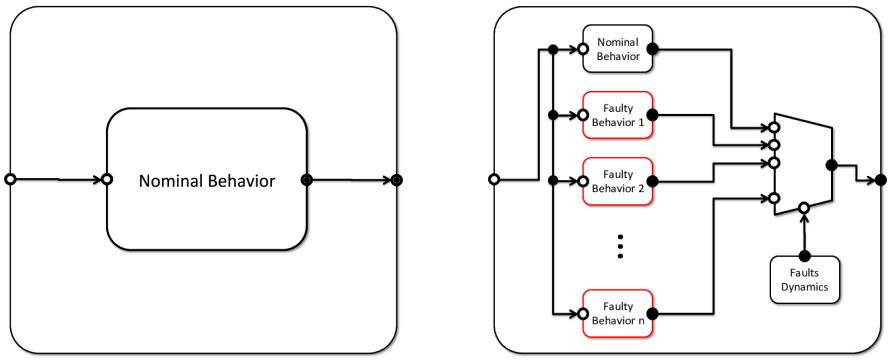
(a) Nominal System (b) Extended System

Figure 5.1: Transition System Extension Example

Definition 5.1.1 (Extended Model of Symbolic Transition System)**.** Given a Symbolic Transition System $S^N = \langle V^N, I^N, T^N \rangle$, called nominal, a Symbolic Transition System $S^X = \langle V^X, I^X, T^X \rangle$ is an *Extended Model* of S^N iff i) $(V^N \cup F) \subseteq V^X$, where F are the fault variables, and ii) for all traces π^N in S^N, there exists a trace π^X in S^X such that $\forall_{f \in F}(\pi^X[i](f) = \bot)$ and $\forall_i \forall_{v \in V^N}(\pi^N[i](v) = \pi^X[i](v))$.

5.2 Fault Injection

The Fault Injection (FI) approach is a model extension technique that guarantees by construction that the nominal behavior, under no failure conditions, is preserved. This approach, developed by the ESACS [42] and ISAAC [11] projects, is directed to extend the model with behavioral faults definitions. This approach assumes the system as combination of components where each of them has input and output ports, and their relation is defined with a behavioral definition. Each component implementing this approach is called *nominal component*, as simply represented



(a) Nominal Component (b) Extended Component

Figure 5.2: Fault Injection Approach

in Figure 5.2a. The application of the fault injection technique, as shown in Figure 5.2b, consists in the definition of an extended component where: 1) each *Faulty Behaviors* (or *Failure Modes*), by preserving the same interface as the nominal one, implements the possible failure effects that deviate from the nominal behavior. The input to the extended component is provided to all behaviors, and 2) the outputs are given to a *multiplexer*. Its output port is then connected to only one of the inputs, and its directly linked to the output of the extended component. 3) The input selection, operated at the multiplexer level, is driven by the *Faults Dynamics* component. Internally it models the transition as a state machine where each behavior represents a state and only the nominal one is initial. The transition to a faulty behavior is triggered by a *fault event*.

The selection of the nominal behavior, in a fault injection setting, implies that the extended component trivially simplifies to the nominal component. Thus, this approach guarantees by construction that the nominal model is preserved.

The extended model definition via Fault Injection allows us to rely on a set of predefined *Failure Modes*. In fact, tools like FSAP [40] and xSAP [27] implement this approach, and provide a set of faulty behaviors like:

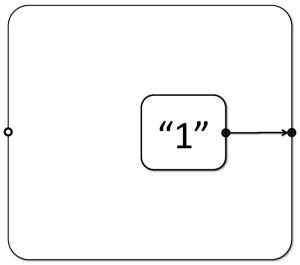


Figure 5.3: Example of a "Stuck at one" faulty behavior.

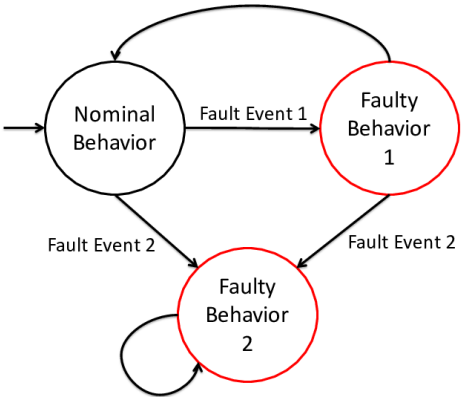


Figure 5.4: Example of a faults dynamics.

- *stuck at zero, one, false,* or *true*: always provides (respectively) the value 0, 1, FALSE, or TRUE as output (an example is shown Figure 5.3);
- *inverted*: it inverts the input, it is equivalent to the logical gate not;
- *random*: it gives as output a non deterministic value in a given range.

Figure 5.4 shows an example of *Faults Dynamics*, in this case the *Faulty Behavior 1* represents a *transient* fault, because it can be recovered back to the *Nominal Behavior*. Moreover, being in *Faulty Behavior 1* or *Nominal Behavior* states, the triggering of *Fault Event 2* enables the *Faulty Behavior 2*. This latter effect is a *permanent* fault, because there is no transition that allows the system to go back to the *Nominal Behavior*.

The Fault Injection approach is tailored for being applied in a completely automated fashion. In fact, in the case of FSAP the inputs to this procedure are i) the nominal model, ii) a list of symbols or variables that are affected by specific failures, and iii) the faults dynamics for each symbol. The latter two inputs are called *Faults Extension Information*. The separation between nominal and extended model is a clear advantage of this technique, considering that different extensions to the same nominal model can be applied only by changing the Faults Extension Information.

However, Fault Injection allows us only to describe failures that affects the internal behavior of a component, which might be limiting when modeling failure effects with different nature. An example of such kind of behavior is a new unintentional interactions between components, like short circuits, which turns out to be impossible to be described with Fault Injection. In addition to that, the system modeling that FI assumes does not fit with a more abstract system definition e.g., requirement based modeling. In fact, if the nominal behavior is defined as a constraint over a set of variables the FI will fail of extending it, because it is meant to affects a specific variable with a specific and localized failure effect.

5.3 Manual Extension

Automated techniques have the advantages of building the extended model upon the nominal one, and guiding the process to reduce design flaws. However, as described in the previous Section, those techniques might be not expressive enough to formalize some kind of system failures.

The most general approach consists in extending the nominal model by explicitly define the faulty behavior. This approach clearly does not impose any boundaries on the failure modeling, a part from the expressiveness of the formalism itself. However, this approach requires more effort, than fault injection, to guarantee that the model extension did not affect the nominal behavior i.e., when no failure occurs the two models are trace equivalent. More specifically, every trace of the nominal model must be a trace also for the extended model.

However, some specific patterns of (manual) model extension are still able to guarantee by construction not to affect the nominal behavior. For instance, the nominal model in SMV language shown in Figure 5.5 represents part of the formalization of an Air Traffic Control System (ATC). The

```
1 MODULE AIR_TRAFFIC_CONTROL(intent_ac_1, intent_ac_2, intent_ac_3)
2 ...
3
4 INVAR
5   ALL_DIFFERENT(sugg_ac_1, sugg_ac_2, sugg_ac_3, ...);
6
7 ...
```

Figure 5.5: Nominal Model Example

```
1 MODULE AIR_TRAFFIC_CONTROL(intent_ac_1, intent_ac_2, intent_ac_3)
2 ...
3
4 VAR
5   resolution_failure : boolean;
6
7 ...
8
9 INVAR
10  !resolution_failure -> ALL_DIFFERENT(sugg_ac_1, sugg_ac_2, sugg_ac_3, ...);
11
12 ...
```

Figure 5.6: Manual Extension Example

aircraft send the intended routes, as trajectory intents (intent_ac_1/2/3 in the example), to the ATC. Those intentions are taken into account for defining the response of the ATC (sugg_ac_1/2/3 in the example), which will confirm or reject the requests by suggesting a new route/trajectory. Such suggested trajectories have to be consistent with a set of constraints, and one of them is that all suggestions must be different for all aircraft in order to avoid collisions (line 5 in Figure 5.5). However, the Air Traffic Control might experience problems when resolving the constraints on suggested routes. This fault effects can be modeled as in Figure 5.6, by defining a new variable that traces the fault occurrence (resolution_failure in the example), and conditioning the constraint to hold only when such variable is assigned to FALSE. Once the fault variables are free to be set to FALSE, this model extension approach guarantees to preserve the nominal behavior.

6

Symbolic Fault Tree Analysis

6.1 BDD-based techniques

The symbolic Minimal Cutsets computation is at the core of all techniques presented in this Thesis, and the BDD-based algorithms described in [37] represent the starting point of the successive SAT-based improvements. This Chapter provides an overview of the symbolic techniques algorithms for performing fault tree analysis, as minimal cutsets computation, using BDDs.

6.1.1 Forward Pruning

The Forward Pruning algorithms is based on a reachability analysis on the symbolic transition system. This approach, described in Algorithm 1, takes as input a model $\mathcal{M}$ (represented as a symbolic transition system with initial state $\mathcal{I}$ and transition relation $\mathcal{T}$), a Top Level Event TLE, and a set of faults F. The result of this algorithm is a formula over F representing the Minimal Cutsets. As described in Chapter 4, a faults combination FC is a cutset if and only if there exists a trace that triggers all faults of FC and it is a counterexample of TLE. The temporal occurrence of the faults in FC is irrelevant, and all their combinations are considered valid.

In order to cover this aspect, the Forward Pruning algorithm defines a monitor of_i (meaning "once f_i") for each fault variable f_i, and all of them are collected into the transition relation $\mathcal{T}^o$ (line 1 of Algorithm 1). The model $\mathcal{M}$ is then extended in line 2 by substituting the transition relation $\mathcal{T}$ with the synchronous products between $\mathcal{T}$ and $\mathcal{T}^o$. The rest of the algorithm, without the *Pruning* optimization, is pretty straightforward. In fact, once the set of the reachable states is computed a projection (line 14) is performed over the once variables. The result of this operation is the set of Cutsets. The last operation maps back the once variables to the actual faults variables.

The pruning approach, defined by the *Pruning* condition in Algorithm 1, consists in optimizing the computation of the frontier (i.e., the new reachable states discovered with one transition relation unrolling) by considering the nature of the CS. More specifically, the Widen function collects all the states that include any element in CS as a proper subset. This needs to update CS in each iteration of the forward image computation, and in general allows for a significant reduction of the search space.

6.1.2 Backward with Dynamic COI

Another approach presented in [37] is the one called Backward with dynamic COI. This techniques builds upon a standard backward reachability that resembles the basic forward search without pruning, in fact it first builds the set of all reachable states, and afterwards projects and minimize CS. The difference of applying a backward search consists in using *guess* variables instead of *once* (they are actually dual of each other).

The Backward with Dynamic Cone Of Influence algorithm is presented in 2. In symbolic minimal cutsets computation, the TLE might be defined over a subset of the variables of $\mathcal{M}$, thus covering only part of the transition relation. In this case the DCOI is able to prune the backward image

Algorithm 1: MCS-BDD Forward-Pruning

Input: Model ($\mathcal{M} = \langle \mathcal{I}, \mathcal{T} \rangle$), Top level event (TLE), Faults (F)
Result: MCS

```
1  T^o := Once(F);
2  M := Extend(M, T^o);
3  Reach := I ∩ (∀_i(of_i = f_i));
4  Front := I ∩ (∀_i(of_i = f_i));
5  while Front ≠ ∅ do
6      if Pruning then
7          CS_o := CS_o ∪ Proj(o, Reach ∩ TLE);          // With pruning
8      Tmp := Reach;
9      Reach := Reach ∪ fwd_img(M, Front);
10     Front := Reach \ Tmp;
11     if Pruning then
12         Front := Front \ Widen(CS_o);                 // With pruning
13 if ¬ Pruning then
14     CS_o := Proj(o, Reach ∩ TLE);                     // Without pruning
15 MCS_o := Minimize(CS_o);
16 return Map_{o→f} (MCS_o)
```

construction in order to consider only the variables that relates with the TLE. This operation is performed by the dcoi_get routine in line 8.

Forward search with pruning and Backward with Dynamic COI are the most efficient BDD-based techniques for Minimal Cutsets computation. The advantage of one compared to the other relates to the nature of the TLE. In fact, backward with DCOI approach is preferable when the TLE is defined over a (significant small) subset of the model variables, for all the other cases the forward search performs better.

Algorithm 2: MCS-BDD Backward-DCOI

Input: Model ($\mathcal{M} = \langle I, T \rangle$), Top level event (TLE), Faults (F)
Result: MCS

```
1  T^g := Guess(F);
2  M := Extend(M, T^g);
3  Reach := TLE ∩ (g = f);
4  Front := TLE ∩ (g = f);
5  i := 0;
6  while Front ≠ ∅ do
7      Tmp := Reach;
8      M^i := dcoi_get(M, TLE, i);
9      Reach := Reach ∪ bwd_img(M^i, Front);
10     Front := Reach \ Tmp;
11     i := i + 1;
12 CS_g := Proj(g, Reach ∩ I);
13 MCS_g := Minimize(CS_g);
14 return Map_{g→f} (MCS_g)
```

6.2 Computing Fault Trees Probability

Particularly important in Safety Assessment is the evaluation of the probability of leading to an undesired condition. This computation can be performed from the results of the Fault Tree Analysis i.e., the Fault Tree. In particular, given a Fault Tree and a mapping $\mathcal{P}$ that links each basic event to its probability of occurrence, it is possible to compute the overall probability of leading to the top-level event. In general, that quantity expresses the probability that the event occurs in a time span of 1 hour of operation, assuming a uniform distribution. Each event is considered independent from the others i.e., two events A and B are independent iff $P(A \cap B) = P(A)P(B)$. Specific techniques for the case of common cause analysis are not considered in this work.

According to those assumptions, representing a Fault Tree as a set of

Algorithm 3: Probability computation.

Input: BDD (n), Probability map ($\mathcal{P}$), Hashtable (*cache*)
Result: Probability

```
1  if n in cache then
2      return cache[n];
3  if n = ⊤ then
4      return 1.0;
5  if n = ⊥ then
6      return 0.0;
7  pthen = Probability_computation(get_then_node(n), P, cache);
8  pelse = Probability_computation(get_else_node(n), P, cache);
9  pcur = P(get_var(n));
10 cache[n] = pcur · pthen + (1.0 − pcur) · pelse;
11 return cache[n];
```

Minimal Cutsets MCS (defined in Chapter 4), the probability of a single fault configuration $FC \in MCS$ is given by the product of the probabilities of its basic faults:

$$\mathcal{P}(FC) = \prod_{f \in FC} \mathcal{P}(f).$$

For a set of minimal cutsets $S = S_1 \cup S_2$, the probability can be computed using the above and the following recursive formula:

$$\mathcal{P}(S_1 \cup S_2) = \mathcal{P}(S_1) + \mathcal{P}(S_2) - \mathcal{P}(S_1 \cap S_2).$$

Implementation of the probability computation

Interpreting the Fault Tree as a propositional formula, and representing it as Binary Decision Diagram, is a simple and efficient way of computing its probability. The algorithm, shown in 3, exploits the following facts:

(i) the probability of two disjoint sets is simply the sum of the two probabilities; and

(ii) the two children t and e of a BDD node with variable v correspond to the two disjoint sets of assignments for the formulae $v \wedge t$ and $\neg v \wedge e$ respectively;

(iii) if the variable v does not occur in the formula f, then f is independent from v, and so $\mathcal{P}(v \wedge f) = \mathcal{P}(v) \cdot \mathcal{P}(f)$;

(iv) $\mathcal{P}(\neg v) = 1 - \mathcal{P}(v)$ by definition.

7

Adaptation of Existing Techniques

The BDD-based approach for formal verification suffers from scalability issues when increasing the state space. The BDD-based Minimal Cutsets computation is based on the same techniques, thus it exhibits the same drawbacks. The advances in symbolic model checking by relying on SAT-based techniques allowed to overcome those limitations. In this Chapter we analyze the existing techniques that can be exploited in order to reach better performance, both by integrating the BMC-based algorithms, and reducing the MCS computation to parameter synthesis.

7.1 Exploiting BMC

An improved version of the BDD-based routines is presented in [32], by exploiting Bounded Model Checking (BMC) as a preprocessing step. Essentially, the idea is to run BMC up to a maximum (user-defined) depth k to check the invariant property stating that the top level event can never be reached. Whenever a counterexample trace is found, a cut set cs (not necessarily minimal) is extracted from it, and the model is strengthened with constraints excluding all the supersets of cs. When no more counterexamples of length (at most) k are found, a BDD-based algorithm is invoked on the strengthened model, in order to discover the remaining cut

sets not yet covered.

The approach can be generalized to completely avoid the use of BDDs. The idea is to use the BMC engine incrementally to enumerate cut sets, and combine it with a generic "black box" procedure for checking invariant properties, invoked periodically (e.g. before increasing the BMC bound k) to check whether all the MCSs have been enumerated.

7.2 MCS via parameter synthesis

The work in [55] presents an efficient extension of the IC3 algorithm (called ParamIC3) that allows us to compute, given a model M depending on some parameters P, the set of all values of P such that the model satisfies a given invariant property. The algorithm works by complement, constructing the set of "good" parameters by incrementally blocking "bad" assignments extracted from counterexample traces generated by IC3.

The technique can be immediately exploited also for MCS computation as follows. First, the model is extended with history variables for fault events, as in the BDD case. The parameter synthesis algorithm is then invoked on the extended model, considering the history variables as parameters, and checking the property that the top level event is never triggered. Each "bad" assignment blocked by ParamIC3 (see [55]) corresponds to a fault configuration reaching the top level event. When the algorithm terminates, the MCS set can be extracted by simply dropping the subsumed bad assignments.

8

Efficient Anytime Techniques

The practical application of model-based safety assessment in an industrial setting poses two key problems. The first one is scalability. In addition to the sheer size of the models, a specific factor is the possibly huge number of relevant MCSs, corresponding to different fault combinations. The second problem is to support the state of the practice. In manual safety analysis, the exploration often proceeds according to the importance and likelihood of fault configurations: MCSs of lower cardinality, that are typically associated with higher probability, are explored before the ones with higher cardinality. When the analysis is considered to be sufficiently thorough, over-approximation techniques are used to assess the weight of the unexplored MCSs.

The work on IC3-based parameter synthesis [55] can in principle address the problem of Minimal Cutsets computation. Here we propose several enhancements based on the specific features of the problem, with dramatic improvements in terms of scalability.

In this Chapter, we investigate and evaluate a family of efficient algorithms for safety analysis. We work under the *monotonicity assumption*, commonly adopted in safety analysis, that an additional fault can not prevent the violation of the property. We specialize IC3-based routines for

parameter synthesis by optimizing the generalization of counterexamples, and by ordering the exploration of MCSs based on increasing cardinality. We also propose a way to accelerate convergence by exploiting the inductive invariants built by IC3.

The practical applicability of our approach is enhanced by proposing a method to precisely compute the under- and over-approximated probability of failure. This technique produces an increasingly precise estimation as the discovery of MCSs proceeds, with the advantage of providing an "anytime" algorithm.

8.1 Efficient algorithms for MCS computation

In practice, the BDD-based routines described in Chapter 6 show rather poor scalability, and are typically not applicable to problems of realistic size. Using BMC as a preprocessing step helps significantly, but ultimately also this technique is limited by the scalability problems of BDD-based approaches. The technique of [55], being based on the very-efficient IC3 algorithm, is much more promising. However, in the basic formulation given in the previous Chapter, its performance is extremely poor when the number of possible fault configurations leading to the top level event is large. In this Chapter, we show how the situation can be dramatically improved by exploiting the monotonicity assumption on faults under which we are operating.

8.1.1 Monotonic parameter synthesis

The first (trivial) improvement exploits the definition of monotonicity to generalize the set of "bad" parameters to be blocked whenever IC3 generates a counterexample trace. This idea is similar to the *dynamic pruning* optimization for the BDD-based computation. The monotonicity assump-

tion ensures that if a set of faults F is sufficient to generate the top level event, so does any set $S \supseteq F$. Therefore, any assignment to the (parameters corresponding to the) fault variables $\gamma = \{f_j, \ldots, f_k\} \cup \{\neg f_i, \ldots, \neg f_h\}$ extracted from an IC3 counterexample trace can be immediately generalized to $\gamma' = \{f_j, \ldots, f_k\}$, by dropping all the variables assigned to false.

The above optimization prevents the algorithm from explicitly considering all cut sets that are subsumed by the one just found, i.e. $F = \{f_j, \ldots, f_k\}$. However, F itself might not be minimal. In this case, IC3 would later have to find another configuration $G \subset F$, and the effort spent in blocking F would have been wasted.

We address this by modifying the branching heuristic of the SAT solver used by IC3. In the modified heuristic, (SAT variables corresponding to) faults are initially assigned to false, and they have higher priority than the other variables, so that no other variable is assigned by a SAT decision before all the fault variables are assigned. This ensures that fault variables are assigned to true only when necessary to satisfy the constraints.

The above idea is very simple to implement and integrate in the IC3-based algorithm (in total, it requires about 20 lines of code), and it provides a significant performance boost (as we will show in Chapter 9). However, it is still not sufficient to ensure that no redundant cut sets are generated. The reason is that, by the nature of IC3, ParamIC3 enumerates counterexample traces in an increasing order of length k, so that it only considers traces of length $k+1$ when all the traces of length $\leq k$ have already been excluded.[1] This means that, if the shortest trace that leads to the top level event from a set F of faults is k, but there exists another set of faults $S \supset F$ that leads to the top level event in $h < k$ steps, then S will necessarily be blocked by

[1]For readers familiar with IC3, strictly speaking this is not fully accurate: if the IC3 implementation uses a priority queue for managing counterexamples to induction [43], some counterexamples of length $h > k$ may be generated before all those of length $\leq k$ are blocked. However, the argument still holds in this case, so the issue can be ignored for simplicity.

ParamIC3 before F. In some extreme cases, this might make the heuristic completely ineffective.

8.1.2 Enumerating only MCS

Algorithm 4: Basic MCS enumeration with ParamIC3

Input: Model ($\mathcal{M} = \langle I, T \rangle$), Top level event (TLE), Faults (F)
Result: MCS

```
1  bound = 1;
2  MCS = ⊥;
3  while True do
4      c = make_atmost(F, bound);
5      region = ParamIC3(I ∧ ¬ MCS, T ∧ ¬ MCS, (¬TLE ∨ ¬c), F);
6      MCS = MCS ∨ ¬ region;
7      done = IC3(I ∧ ¬ MCS, T ∧ ¬ MCS, ¬TLE);
8      if done then
9          return MCS
10     else
11         bound = bound + 1;
```

We address the problem by incorporating in our algorithm a solution originally proposed in [3]. The idea is to force the algorithm to proceed by *layering*, by forcing the search to compute the cuts sets of increasing cardinality, instead of analyzing traces of increasing length. The pseudo-code for the basic version is shown in Algorithm 4. At each iteration of the main loop, the algorithm uses an "atmost" constraint c to limit the cardinality of the cut sets generated, by relaxing the invariant property to check from ¬TLE to $(\neg\text{TLE} \vee \neg c)$. The termination check is performed by invoking the "regular" version of IC3 on the model strengthened to exclude the already-computed cut sets, to check whether there are other fault configurations that can reach the top level event. It is easy to see that Algorithm 4 enumerates only the MCSs, and thus it avoids the exponential

blow-up suffered from ParamIC3 on the model of Example 8.1.1. However, it does so at a significant price, since it needs two IC3 calls per iteration. On less pathological examples, the overhead introduced might largely outweigh the potential benefits.

Algorithm 4 can be improved by exploiting the capability of IC3 (and so also of ParamIC3) of generating a proof for verified properties in the form of an *inductive invariant* entailing the property P. In our specific case, the inductive invariant ψ produced by ParamIC3 on line 5 of Algorithm 4 would satisfy the following: (i) $I \wedge \neg\text{MCS} \wedge \text{region} \models \psi$; (ii) $\psi \wedge T \wedge \neg\text{MCS} \wedge \text{region} \models \psi'$; and (iii) $\psi \wedge \neg\text{MCS} \wedge \text{region} \models (\neg\text{TLE} \vee \neg c)$. The first improvement is based on the observation that the inductive invariant can be fed back to ParamIC3 at the next iteration of the main loop, thus avoiding the need of restarting the search from scratch. The second improvement, instead, exploits the computed invariant to check whether all the MCSs have been enumerated, thus avoiding the second invocation of IC3 of line 7. This is done by checking with a SAT solver whether the current invariant ψ is strong enough to prove that the top level event cannot be reached by any fault configuration not covered by the already-computed cut sets. Note that this does not affect completeness, since in the worst case the atmost constraints simplifies to true after $|F|$ iterations of the loop. However, the hope is that in practice the inductive invariant will allow us to exit the loop much earlier. The enhanced algorithm is shown in Algorithm 5, where the improvements are displayed in red.

Example 8.1.1. Consider the following example, using the syntax of nuXmv.

```
1 MODULE main
2 IVAR
3   fault_1 : boolean;
4   ...
5   fault_N : boolean;
6
7 DEFINE fault_count := fault_1 + ... + fault_N;
8
9 VAR counter : 1 .. N;
10     status : boolean;
11
12 ASSIGN
13   init(counter) := 1;
14   next(counter) := counter = N ? 1 : counter + 1;
15
16 TRANS (fault_count = 0) | (fault_count > (N - counter));
17
18 ASSIGN
19   init(status) := TRUE;
20   next(status) := (fault_count = 0);
```

There are N fault variables, and suppose the top level event occurs when the status variable becomes false, i.e., whenever at least one fault occurs. Therefore, the MCSs for this model are the N singleton sets containing one fault variable each. However, the TRANS constraint forces an inverse dependency between the number of steps to reach the top level event and the cardinality of the smallest cut sets needed: for k steps, the smallest cut sets have cardinality $N - k$, and there are $\binom{N}{k}$ of them. Therefore, even with the branching heuristic described above, ParamIC3 will generate an exponential number of counterexamples (since $\sum_{k=1}^{N} \binom{N}{k} = 2^N - 1$) before finding the MCSs. ◇

8.2 Anytime approximation

An additional benefit of Algorithm 5 compared to the other algorithms is that it provides an "anytime" approximation behaviour on the set of MCSs, in the sense that at any point during its execution, the candidate solution is a subset of all the MCSs. As pointed out in Chapter 2, however, such underapproximation is useful only if it is possible to estimate its error in terms of failure probability. Here, we show a simple but effective procedure for estimating the approximation error on the fly, during the execution

Algorithm 5: Enhanced MCS enumeration with ParamIC3

Input: Model ($\mathcal{M} = \langle I, T \rangle$), Top level event (TLE), Faults (F)
Result: MCS

```
1  bound = 1;
2  MCS = ⊥;
3  invar = ⊤;
4  while True do
5      c = make_atmost(F, bound);
6      region, invar = ParamIC3(I ∧ ¬ MCS ∧ invar, T ∧ ¬ MCS ∧ invar,
                                (¬TLE ∨ ¬c), F);
7      MCS = MCS ∨ ¬ region;
8      done = check_unsat(¬ MCS ∧ invar ∧ TLE);
9      if done then
10         return MCS
11     else
12         bound = bound + 1;
```

of Algorithm 5. This allows us to consider a bound on the error as an alternative stopping criterion for the algorithm, which might be useful in cases when the full computation of all the MCSs would be too expensive.

The idea is to keep two running bounds for the probability x of reaching the top-level event, such that at any point in the execution of the algorithm $P_L(\text{TLE}) \leq x \leq P_U(\text{TLE})$. Initially, we set $P_L(\text{TLE}) = 0$ and $P_U(\text{TLE}) = 1$. When a minimal cut set m_1 is found, $P_L(TLE)$ is incremented by computing the probability of the fault configurations represented by m_1 that are not covered by the already-computed MCSs. This can be done by constructing the BDD for the formula $m_1 \wedge \neg\text{MCS}$, and then computing its probability with Algorithm 3.[2]

For updating the upper bound $P_U(\text{TLE})$, instead, we exploit fact that Algorithm 5 proceeds by layers of increasing cardinality. More precisely,

[2]For performance reasons, it might make sense to perform this computation for clusters of cut sets rather than for individual ones, trading granularity for efficiency.

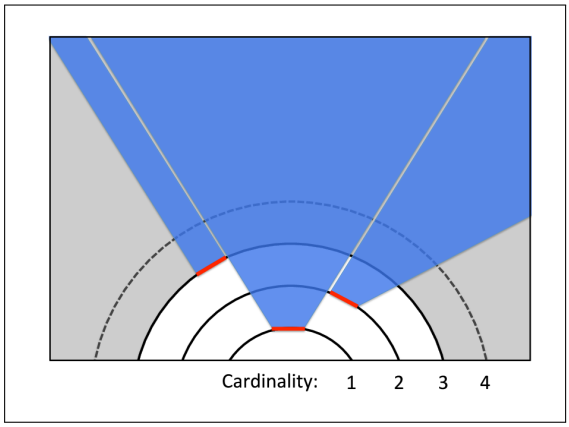


Figure 8.1: Illustration of the probability error estimation in Algorithm 5.

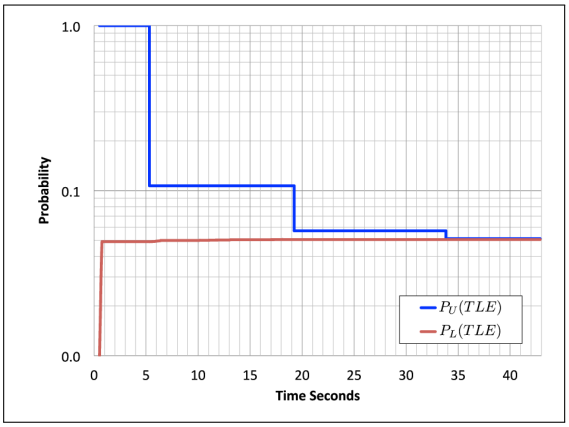


Figure 8.2: Example of evolution of probability error bounds.

when ParamIC3 returns at line 7, we know that all the fault configurations of cardinality smaller or equal to the current bound that are not included in MCS will definitely not cause the top-level event. The probability P_{excluded} of these configurations can be computed with Algorithm 3 by constructing the BDD for the formula $\neg\text{make\_atmost}(\vec{F}, \text{bound}) \wedge \neg\text{MCS}$. With this, the new value of $P_U(\text{TLE})$ is given by $1 - P_{\text{excluded}}$. An illustration of this idea is shown in Figure 8.1. The red area represents the minimal cut sets found within a specific cardinality, and the blue one shows all the supersets of those cut sets. The white area denotes the configurations that cannot cause the TLE, whereas the gray one represents the unknown part. Figure 8.2 shows instead an example of the evolution of the error bounds during the execution of Algorithm 5 for one instance of our benchmark set: $P_L(\text{TLE})$ becomes non-zero after the first cut set found, and then grows continuously at every cut set, whereas $P_U(\text{TLE})$ decreases in steps, whenever an individual cardinality has been fully explored.

9

Experimental Evaluation

We have implemented the algorithms described in the previous sections in the xSAP, which is model-based safety analysis platform. In this Section, we experimentally evaluate their performance and effectiveness.

9.1 Benchmarks

The benchmarks used for the evaluation come from a set of real-world test cases from the avionics domain, where safety assessment and Fault Tree Analysis are parts of the formal analysis of the models.

9.1.1 Aircraft Electrical System.

The first set of benchmarks describes the architecture of an aircraft-oriented electrical system. These problems were developed as part of the MISSA project [94], and previously analyzed using OCAS, a proprietary model-based safety assessment platform, as well as the FSAP [40] toolset. This comparison is described in [32]. This family of benchmarks is composed of four different models, where each of them is a refinement of the previous one. The properties that are taken into account describe the situation when the system that manages the alternate/continuous current is mal-

functioning. Each model has two properties, for a total of 8 benchmark instances. The size of the models varies from 35 to 297 state variables and from 437 to 14030 AND gates (in an And-Inverter-Graph representation [26] of the transition relation), whereas the number of faults is between 9 and 105.

9.1.2 Next-gen collision avoidance.

The second set of instances comes from the analysis of a novel, "next generation" air traffic control system that is being studied at NASA, and described in detail in Chapter 29. Part of the activities involves the evaluation of different technological approaches in order to discover the safer and most efficient one. This process is supported by different analysis techniques, and one of those is based on formal model-based safety assessment. The formal model is composed of an on-ground Air Traffic Control System (ATC), a set of aircraft that rely on ground-based separation systems like the ATC (GSEP), and a set of aircraft that have self-separating capabilities (SSEP) as support of the standard ground-based approach.

The benchmark instances encode different architectural solutions for the Next-gen collision avoidance system. The system is composed of various numbers of GSEP and SSEP aircraft, and one ATC. The models contain 47 basic faults, and the objective is to compute the MCSs for the violation of the property "Two Aircraft shall not have a Loss of Separation", meaning that the distance between two aircraft is below a certain safety limit. The models are scaled by varying the number of aircraft of each kind (GSEP and SSEP, from 0 to 3 each) and the number communication rounds between each aircraft and the ATC (from 1 to 10). The size of the models varies from 162 to 330 state variables and from 1700 to 5110 AND gates.

9.1.3 Wheel Braking System.

The third family of benchmarks models an aircraft-based wheel braking system (WBS), described in the Aerospace Information Report, version 6110 [109]. The model was developed in a joint project between FBK and the Boeing Company [36], and it is representative of an industrial system of significant size. Further details about this case study are discussed in Chapter 31. The WBS describes a redundant architecture that takes as input the pedal information (the brake signal coming from the pilot), computes the braking force that has to be applied to the 8 wheels, and drives the hydraulic system in order to physically operate the right braking force. This system is characterized by three redundant sub units:

(i) normal brake system, receiving the pedal information and driving the hydraulic system. This unit is composed of two sub components that work in parallel in order to prevent that a single failure can cause the complete malfunctioning;

(ii) alternate brake system, receiving the pedal information and the output from the normal brake system: when the latter one is not operating as expected, it operates as backup by driving the hydraulic system;

(iii) emergency brake system, behaving similarly to the alternate one: it receives pedal information and both outputs from the normal and alternate sub systems, and operates as a backup of the alternate one.

The benchmark set consists of 4 different variants of the WBS architecture, expressing various kinds of faulty behaviour. The models contain 261 fault variables and 1482 state variables, whereas the number of AND gates varies between 35182 and 35975.

9.2 Performance evaluation

In the first part of our analysis, we evaluate the performance of different techniques for the computation of the set of MCSs. We consider the following algorithms:

BDD is the procedure of [37] described in Chapter 6;

BMC+BDD is the enhancement of [32] that uses BMC as a preprocessor. The BMC implementation uses the branching heuristic described in Section 8.1 for reducing the number of fault configurations to enumerate;

BMC+IC3 is the variant of the previous technique outlined in Chapter 6, using IC3 as a "black box" invariant checking procedure. (The branching heuristic of Section 8.1 for fault variables is used also in this case);

ParamIC3 is a basic version of ParamIC3, exploiting monotonicity for generalizing parameter regions to block;

ParamIC3+faultbranch is the enhanced version of ParamIC3 that uses the branching heuristic for fault variables of Section 8.1;

MCS-ParamIC3-simple is the basic MCS procedure described in of Algorithm 4. We use m-cardinality networks [4] for encoding the cardinality constraints;

MCS-ParamIC3 is the enhanced MCS procedure of Algorithm 5;

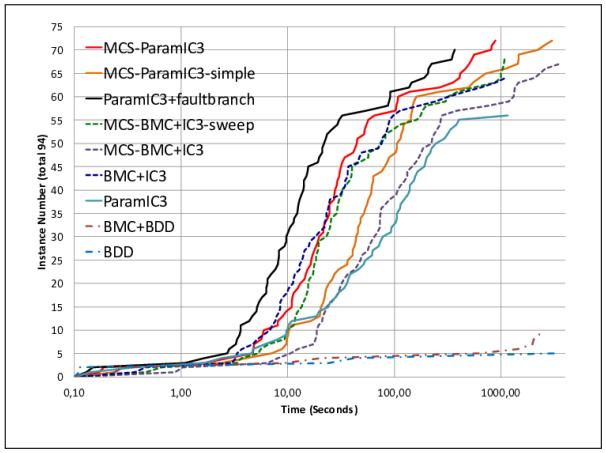
MCS-BMC+IC3 is an anytime variant of BMC+IC3, in which the BMC solver is forced to enumerate only MCSs, using cardinality constraints: whenever IC3 finds that a given cardinality has been fully enumerated, the bound of the atmost constraint is increased, and BMC is restarted;

MCS-BMC+IC3-sweep is a variant of the above, in which IC3 is invoked less frequently and BMC is limited to a maximum counterexample length k, instead of fully enumerating a given cardinality. This is expected to improve performance, at the price of losing the "anytime" feature. Intuitively, instead of analyzing the possible dimensions in sequence (i.e., first consuming all MCS cardinalities, then all BMC k), it proceeds "diagonally" by alternating them.

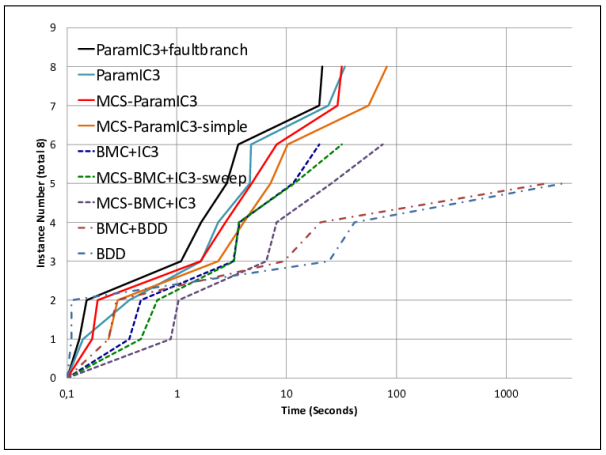
We have run our experiments on a cluster of Linux machines with 2.5GHz Intel Xeon E5420 CPUs, using a timeout of 1 hour and a memory limit of 4Gb. The results are shown in Figure 9.1. The plots show the number of solved instances (y-axis) in the given timeout (x-axis) for each of the algorithms considered. More information is provided in Table 9.1, where for each configuration we show the number of solved instances and the total execution time (excluding timeouts).

From the results, we can clearly see the benefits of the techniques discussed in Section 8.1. Using the specialized branching heuristic, ParamIC3+faultbranch performs very well in general, especially on the Elec.Sys and NextGen families. However, for the harder WBS instances, the heuristic is not enough. On the contrary, the cardinality-based enumeration introduces an overhead for easier problems, but it pays off for harder ones, making MCS-ParamIC3 the best performing overall. Moreover, even for simpler problems the integrated approach of Algorithm 5 is not very far from the performance of ParamIC3+faultbranch. More importantly, the anytime behaviour of MCS-ParamIC3 is extremely useful in all cases in which none of the algorithms terminates, i.e. in the majority of the WBS instances. Its usefulness is evaluated in Section 9.3.

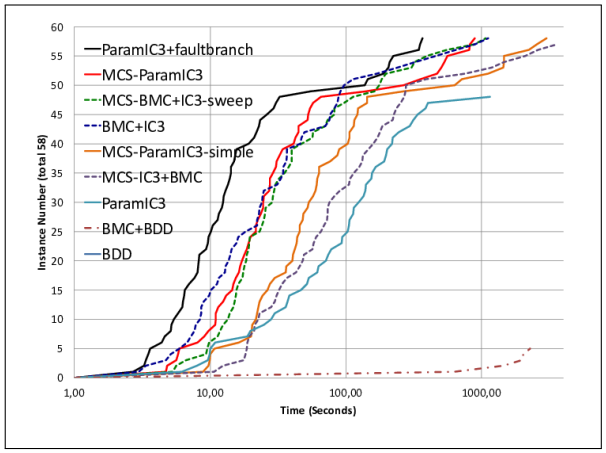
All instances.



Elec.Sys instances.



NextGen instances.



WBS instances.

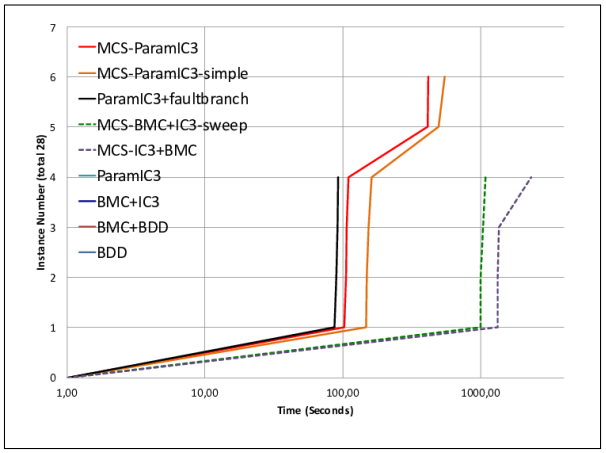


Figure 9.1: Results of performance evaluation.

9.3 Error estimation

In order to assess the usefulness of the anytime behaviour, we evaluate the effectiveness of our technique for estimating error bounds on the probability of faults. For this, we consider the instances of the WBS benchmark set that could not be completed within the timeout, and for each of them we study the evolution of the probability bounds during the execution of MCS-ParamIC3. The results are summarized in Table 9.2, where we show the number of MCSs found of each cardinality, as well as the evolution of

Table 9.1: Summary of scalability evaluation.

Algorithm	**# solved**				**Total Time (sec)**
	All	**Elec.Sys**	**NextGen**	**WBS**	
MCS-ParamIC3	**72**	8	58	6	7837
MCS-ParamIC3-simple	**72**	8	58	6	19326
ParamIC3+faultbranch	**70**	8	58	4	3222
MCS-BMC+IC3-sweep	**68**	6	58	4	9896
MCS-BMC+IC3	**67**	6	57	4	23210
BMC+IC3	**64**	6	58	0	5477
ParamIC3	**56**	8	48	0	6787
BMC+BDD	**10**	5	5	0	10753
BDD	**5**	5	0	0	3377

the probability bounds during the execution for a representative subset of the WBS instances (we could not include all instances for lack of space).

From the table, we can see how for most instances error bounds converge quickly towards the actual fault probability, and then continue improving very slowly, confirming the intuition of safety engineers that it is often enough to consider only MCSs of small cardinality in practice. There is only one case where the bounds are very loose, namely the M1-S18-WBS-R-323 instance. However, in this case the fault probability is several order of magnitudes smaller than for the other properties.

We remark that the probabilities for the basic faults are not artificially generated; on the contrary, they have been estimated by domain experts, and the error bounds that we have obtained matched their expectations. The table shows that, for these problems, the error estimation provided by our technique is precise enough to make our results useful in practice even when the computation of the set of MCSs does not terminate.

Table 9.2: Evolution of probability error bounds on hard WBS instances.

Instance	**card**	**# MCS**	**Time**	P_L(**TLE**)	P_U(**TLE**)
M1-S18-WBS-R-0321	2	6	3.686	4.4999799997e-10	4.7856862743e-09
	3	627	27.937	4.5052040749e-10	4.5368234398e-10
	4	629	96.760	4.5052047798e-10	4.5052230781e-10
	5*	38950	3549.163	4.5052047798e-10	4.5052230781e-10
M1-S18-WBS-R-0322-left	1	2	1.809	9.9999750001e-06	1.4392898712e-05
	2	2	3.827	1.0000324995e-05	1.0004616980e-05
	3	203	23.106	1.0000325102e-05	1.0000328223e-05
	4*	46287	3271.215	1.0000325102e-05	1.0000328223e-05
M1-S18-WBS-R-0323	6	13689	480.034	1.0696143952e-28	3.5789505917e-22
	7*	52035	3596.097	1.0701599223e-28	3.5789505917e-22
M1-S18-WBS-R-0324	2	1	3.603	2.5000000000e-11	4.3619410877e-09
	4	2	9.273	2.5000000001e-11	2.5001833724e-11
	5	8729	360.012	2.5000000003e-11	2.5000000881e-11
	6*	23995	2905.057	2.5000000003e-11	2.5000000881e-11
M1-cmd_implies_braking_w1	1	13	4.508	1.1299483157e-04	1.1708790375e-04
	2	30	12.944	1.1299924596e-04	1.1300309322e-04
	3	7428	265.771	1.1299925205e-04	1.1299925473e-04
	4	3815	865.818	1.1299925205e-04	1.1299925205e-04
	5	1768	1956.225	1.1299925205e-04	1.1299925205e-04
	6	168	3465.792	1.1299925205e-04	1.1299925205e-04
M2-S18-WBS-R-0321	2	6	3.772	4.4999799997e-10	4.7856862743e-09
	3	1252	47.248	4.5075536751e-10	4.5391665924e-10
	4	629	113.151	4.5075543799e-10	4.5075726695e-10
	5*	30977	3379.841	4.5075543800e-10	4.5075726695e-10
M2-S18-WBS-R-0322-left	1	2	1.855	9.9999750001e-06	1.4392898712e-05
	2	2	4.071	1.0000324995e-05	1.0004616980e-05
	3	732	40.776	1.0000325300e-05	1.0000328420e-05
	4*	47583	3070.011	1.0000325300e-05	1.0000328420e-05
M2-S18-WBS-R-0323	6	13689	470.905	1.0696143952e-28	3.5789505917e-22
	7*	53241	3577.552	1.0701693234e-28	3.5789505917e-22
M2-S18-WBS-R-0324	2	1	3.160	2.5000000000e-11	4.3619410877e-09
	4	38	11.297	2.5000000077e-11	2.5001833799e-11
	5	10859	524.662	2.5000000079e-11	2.5000000958e-11
	6*	26943	3458.824	2.5000000079e-11	2.5000000958e-11

*: cardinalities for which not all the MCSs could be computed within the timeout

10

Top Level Event as LTL property

The definition of system requirements as invariant properties is often too limiting when analyzing a complex system. This requires reliance on techniques and tools implementing the support for richer formal specifications such as Linear Temporal Logic.

The support for full LTL model checking is already implemented in a variety of tools (see Section 4.7), however no techniques has been introduced in order to support, in model-based safety analysis, the definition of top level events as LTL properties. In the formal validation, verification, and safety assessment process, the top level events usually represent a negation of a verification property. "A brake command implies that the aircraft will eventually stop" is a typical fairness LTL property that is supposed to be met in the nominal model, whose negation represent a top level event.

In this Chapter we introduce an extension of the invariant based techniques, discussed earlier in this Part, in order to support fault tree analysis with LTL properties as TLE.

10.1 Infinite Traces

The result of the problem $S \models \varphi$ is either positive, or a witness trace π of S such that $\pi \not\models \varphi$. In case of a safety or invariant property the

trace π can be finite, where its last state is actually violating the invariant condition. Therefore, a trace that does not model a liveness property should be infinite, in fact $\pi = s_0, s_1, \ldots \not\models \mathbf{F}(good)$ only when the condition $\forall_{i \geq 0} s_i \not\models good$ holds. In practice, an infinite trace is usually represented as a finite trace $\pi = s_0, s_1, \ldots, s_n$ where there exists a $0 \geq k < n$ such that $s_n = s_k$. This representation is called *lasso shaped*, due to the fact that the last state of the trace is "bend" to coincide to one in the middle. Thus, a lasso shaped trace $\pi = s_0, s_1, \ldots, s_k, s_{k+1} \ldots, s_n$, where $s_k = s_n$, represents the infinite trace $\pi = s_0, s_1, \ldots, s_k, (s_{k+1} \ldots, s_n)*$ where $s_{k+1} \ldots, s_n$ is repeated infinite many times.

The symbolic fault tree analysis, based on symbolic model checking, can be seen as the problem of finding all faults configurations $fc \in FC$ such that there exists a trace that triggers those faults and not models the TLE property. This applies independently from the fact that the trace is finite of infinite. However, the probability computation over the resulting fault tree might be not correctly represented. This applies when computing the probability of a minimal cutset that appears in the model only in the repeated part of an infinite (lasso shaped) trace. In fact, the actual probability of such cutsets cs is ~ 0 (i.e., $\prod_0^\infty \mathcal{P}(cs) = 0$ given $0 \geq cs < 1$), but it will counted as $\mathcal{P}(cs)$. The resulting computed probability is thus an over approximation of the actual one, which is considered reasonable in safety assessment, because it does not mask relevant faults configurations.

10.2 BMC-based approaches

One possible approach to solve this problem can be based on LTL model checking via Bounded Model Checking [24]. In particular, it is possible to proceed iteratively for each witness trace as a counter example of the model checking, and blocking each fault configuration fc at every step i.e.,

extending the property $\varphi := \varphi \vee \mathbf{F}(fc)$. Practical improvements of this approach integrate the branching heuristics as well as the only enumeration of the minimal cutsets, which are similar to the ones presented in Chapter 8.

The BMC based technique is, however, a "partial verifier" for the LTL model checking problem, meaning that the possible answers are *FALSE* (with a counterexample) and *UNKNOWN* when there is no answer. During the enumeration of the minimal cutsets computation, not reaching a *TRUE* result might hide a trace violating the given property without triggering any fault. In this case, the whole fault tree would collapse into the empty set i.e., every fault configurations are supersets of the empty one.

One possible approach to provide a complete (no *UNKNOWN*) result is to combine the BMC approach with LTL model checking via BDD [66, 116]. This can be done by integrating a BDD check $S \models \mathbf{G}(\neg FT) \rightarrow \varphi$ that evaluates if the BMC was able to find all fault configurations. A positive result will terminate the analysis, while a negative one would require to increase the depth of the BMC evaluation in order to discover more cutsets. The LTL fault tree analysis on finite states systems via the BMC+BDD approach guarantees to terminate i.e., there exists a BMC depth where reachable states are considered.

10.3 Extension with K-liveness

BDD-based techniques are not particularly efficient to deal with large systems due to a blow up in memory consumption. This drawback applies also to the BMC+BDD-based minimal cutsets computation. Thus, a more efficient way of performing LTL fault tree analysis consists in rely on pure SAT-based techniques. The IC3-based K-liveness [60] is one of the most promising approaches to replace the BDD-based check. More specifically, this technique reduces LTL verification to a sequence of invariant checking.

The key insight of K-liveness is that, for finite-state systems, this is equivalent to find a K such that a signal f is visited at most k times, which in turn can be reduced to invariant checking (i.e., it tries to prove $\mathbf{FG}\neg f$). This approach is integrated in similar way of the MCS-BMC+IC3-sweep algorithm described in Section 8.2. We compared this technique with a conceptual implementation that reduces a liveness check into an invariant checking [23], and that it performs afterwards an IC3-based minimal cutsets computation over the resulting model. This analysis showed that the method based on MCS-BMC+IC3-sweep provides better scalability results compared to the liveness-to-safety approach. Further experimental evaluations on these techniques are delegated to future works.

11

Future Extensions

In this Chapter we presented a family of algorithms for model-based safety analysis, based on IC3. The algorithms tightly integrate the generation and minimization of cut sets, and enable the computation of the hazard probability, both numerically and symbolically. Moreover, we introduced a method to provide an estimate for the remaining computation, when the generation does not terminate, and to safely approximate the final result. This makes way for an anytime approach, and provides the possibility to deal with cases where the number of cut sets may explode. In addition to that, the extension that allows us to deal with TLE expressed as LTL, represents a significant improvement over the previous techniques, which support only invariant properties.

An important open challenge we wish to explore is the relaxation of the monotonicity assumption on faults. Traditionally, in the avionics and aerospace domain (from which our benchmarks are taken) non-monotonic analysis is rarely considered, as it does not provide significant benefits – most systems are indeed monotonic and, whenever they are not, monotonic analysis already provides an accurate over-approximation. However, in other domains this is known not to be the case: for example, in circuits two subsequent inversions may prevent the occurrence of a top level event.

Given the hardness of the non-monotonic analysis, it may be also worth to compute a monotonic over-approximation and find other means to tighten the measure (or to compute the tightness of the approximation). Finally, we want to study strategies to detect non-monotonicity, as in some cases it may be unclear whether it holds or not.

Part III

Compositional Safety Analysis

Introduction

Complex systems are often the result of two complementary processes. On the one side, *hierarchical design* refines a set of requirements into increasingly detailed levels, decomposing a system into subsystems, down to basic components. On the other side, the process of *safety assessment* (SA) analyzes the impact of faults, and pinpoints their consequences (e.g., a valve failing to operate) on high-level functions (e.g., loss of thrust to engines).

In architectural design, the failures of components are typically not modeled explicitly, thus they typically artificially introduced in the model for safety assessment. However, the design that is later implemented in real software and hardware components contains only nominal interfaces and behaviors, which may contain redundancy mechanism or failure monitoring, but not the failure themselves. We call such architectural design the *nominal architecture*. Modeling and analysis of faults is the objective of safety assessment, however there is often a gap between the design of the nominal architecture and SA, which are carried out by different teams, possibly on out-of-sync components. This requires substantial effort, and it is often based on unclear semantics.

In this Part we show a new formal methodology to support a tight integration between the architectural design and the safety assessment process [35]. Our approach builds on two main ingredients. First, we use Contract-Based Design (CBD) - a technique that provides formal support to the architectural decomposition of a system into subsystems and sub-

components. Components at different levels of abstraction are characterized by contracts (assumptions/guarantees). CBD can provide feedback in the early stages of the process, by specifying blocks in abstract terms (e.g., in terms of temporal logic [58]), without the need for a behavioral model (e.g., in terms of finite-state machines).

Second, we use the idea of fault injection (a.k.a. model extension), which enables the transformation of a nominal model into one encompassing failures. This is done by introducing additional variables driving faults activation, hence controlling whether the system is behaving according to the nominal or the faulty specification. Within this setting, it is possible to automatically generate Fault Trees (FTs) using model checking techniques. This approach focused in the past on behavioral models [94, 41], with a significant limitation of generating two-levels "flat" FTs, corresponding to the DNF of their minimal cut sets (MCSs) [37]; as such, it is unable to exploit system hierarchy.

The novel contribution of our approach is the extension of CBD for SA (CBSA): given a nominal contract-based system decomposition, we automatically obtain a decomposition with fault injections. The insight is that the failure mode variables are directly extracted from the structure of the nominal description, and that they model the failure of a component to satisfy its contract. The approach is proved to preserve the correctness of refinement: the extension of a correct refinement of nominal contracts yields an extended model where the refinements are still correct. Once the contracts are extended, it is possible to automatically construct FTs that mimic the structure of the architecture, and formally characterize how lower-level or environmental failures may cause failures at higher levels. This approach has several important features. First, it is fully automated, since SA models are directly obtained from the design models, without further human intervention. Second, it can be applied early in the devel-

opment process and stepwise along the refinement of the design, providing a tight connection between design and SA. Third, it allows for the generation of artifacts that are fundamental in SA, namely FTs that follow the hierarchical decomposition of the system architecture.

The framework has been implemented extending the OCRA tool [54], which supports CBD. We show experimentally that our approach is able to produce hierarchically organized FTs automatically and efficiently. Furthermore, when applied to behavioral descriptions, the partitioning provided by CBD demonstrates a much better scalability than the monolithic approach, which does not consider the hierarchical system decomposition.

The rest of Part III is structured as follows:

- **Background**
  - Chapter 12 provides an overview of the current automated techniques for structured fault trees generation;
  - Chapter 13 describes the Wheel Braking System, which is used as running example in the description of this Part;
  - Chapter 14 provides a characterization of the Contract-Based technique;
- **Contributions**
  - Chapter 15 extends the contract-based approach with Safety Assessment.
  - Chapter 16 discusses the results of the experimental evaluations;
  - Chapter 17 concludes with the foreseeable future extensions.

12

Automated Generation of Structured Fault Trees

In recent years, as discussed in Chapter 3, there has been a growing industrial interest in MBSA, see e.g., [41]. These methods are based on a single safety model of a system. Formal verification tools based on model checking have been extended to automate the generation of artifacts such as FTs and FMEA tables [41, 40, 31, 37, 32], and used for certification of safety critical systems, see e.g., the Cecilia OCAS platform by Dassault Aviation. However, the scope of such methods is limited to the generation of the MCSs, represented as a two-level FT. This limitation has an impact in terms of scalability, and readability of the FTs. The approach described in this Part overcomes those limitations – both in terms of scalability and significance of the generated FTs (we produce hierarchically organized FTs, as per [107]). Moreover, as a difference with traditional MBSA, we follow a fully top-down development approach, which closely resembles the SA process as described, e.g., in [107], providing feedback in much earlier stages of the design.

An alternative approach for the generation of more structured FTs is based on actors-oriented design [101, 91], however these techniques do not account for a stepwise refinement of SA, as outlined in [107]. Specifically,

even in presence of minor changes, this approach does not provide the possibility to refine, extend or reuse previous FTA.

Different approaches tackle the problem of automated construction of safety artifacts by inspecting SysML and UML system modelings. In [99, 85], Fault Trees are synthesized from UML diagrams by applying a model transformation technique. A different approach described in [75] aims at providing a set of formal analysis, that rely on Matlab/Simulink/Stateflow, in order to support the safety analysis process and increase the confidence of the produced results. A different approach that aims at automatizing the safety analysis is proposed in [28]. In particular, it applies a transformation from UML to PROMELA (the formal language of the SPIN [73] model checker) for the formal verification part, and to a subset of Stochastic Petri Nets for the quantitative analysis. However, the Fault Tree generation is carried out by inspecting predefined architectural patterns.

Our work is similar in spirit to [15], which presents a methodology based on retrenchment (an extension of classical refinement), to generate hierarchical FTs from systems represented as circuits, exploiting the system dataflow. A major difference is that retrenchment does not focus on top-down development, but rather on the relation between nominal and faulty behaviors. It takes as input the system hierarchy and the behavioral models, hence it does not support the FT generation along the stepwise refinement. Moreover, the framework is theoretical and, although an algorithm for generation of FTs is provided, implementation issues for its realization are not discussed.

In [16], contracts are (manually) generated after a safety and design process. The FT is manually constructed starting from some diagrams describing the system behavior. State machines are extended with faulty behavior to analyze the hazards. Differently from our work, FTA and hazards analysis are used to collect information to specify the contracts.

We instead start from the contracts to derive automatically the FT.

In this work, we based the fault-tree generation on the contract-based refinement. There are other more mature refinement techniques such as the B Method [6], but we are not aware of approaches to FT generation based on these refinements.

Finally, in the context of fault diagnosis, the work described in [112] constructs diagnoses by exploiting the hierarchy of a circuit; the health variable associated with a region of the circuit, called cone, resembles the idea of intermediate event in a FT. However, this work does not focus on architectural design and stepwise refinement.

13

The Wheel Braking System Example

In this work, we integrate the formal description of the Contract-Based Safety Analysis technique with an explanatory running example: the Wheel Braking System (WBS). This case study was introduced in [107], and later used to describe a formal specification ([64]) and refinement ([58]) of contracts along a system architecture – it is therefore an ideal case study to evaluate our approach. The Wheel Braking System controls the braking of the main gear wheels for taxiing and landing phases of an aircraft. The architecture of the WBS is organized in multiple levels, and each of them refines the previous one:

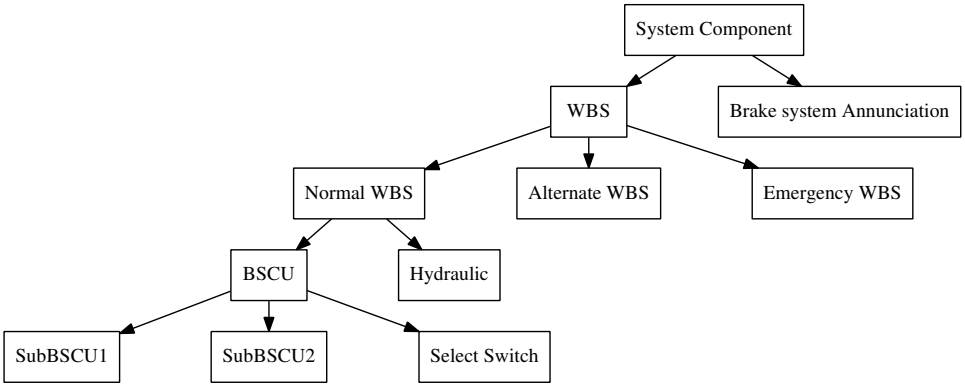


Figure 13.1 shows the WBS architecture, where the top level component is called *System Component*, and contains the whole architecture. This component is then refined with the *Wheel Braking System* (WBS),

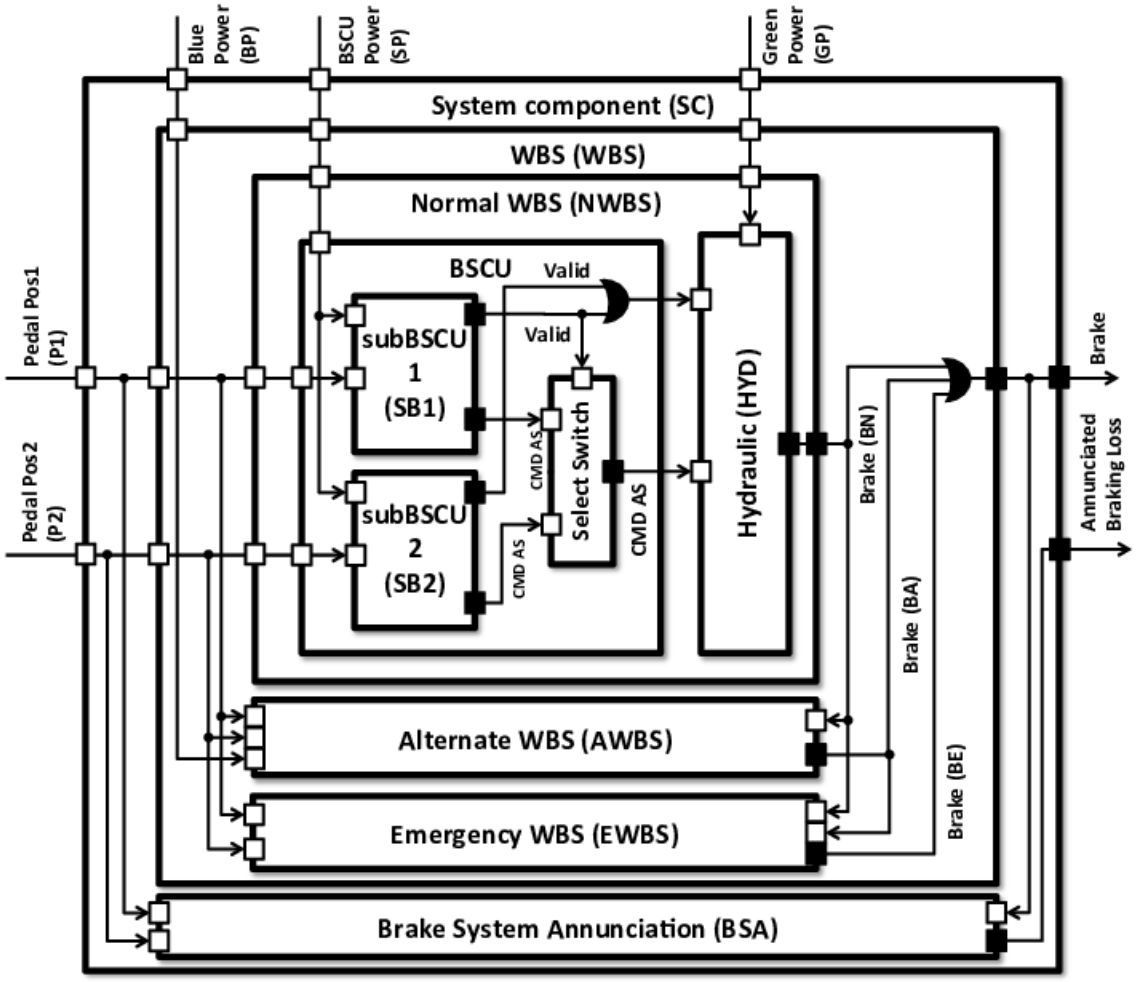


Figure 13.1: WBS architecture (the names in parenthesis define the abbreviations)

which is the actual braking system, and the *Braking System Annunciation* (BSA) that operates as a monitor on the WBS activity, raising a signal in case of the WBS fails to operate. The WBS can be commanded either automatically or manually via brake pedals, and it is composed of three sub systems: *Normal* (NWBS), *Alternate* (AWBS), and *Emergency* (EWBS). Those systems operate in cascade, in fact the alternate system is in stand by and is selected automatically when the normal one fails. The emergency brake system is activated when both NWBS and AWBS fail. In normal mode, the brake is controlled by the *Braking System Control Unit* (BSCU),

implemented with two redundant control units. Each sub-unit (SB1 and SB2) receives an independent pedal-position signal. Monitors detect the failure of the sub-units, producing the "Valid" signals, and of the whole BSCU. Moreover, the Wheel Braking System relies on hydraulic power to perform the brake operations. This system is redundant, and it is based on two independent sets of hydraulic pistons, supplied by independent power lines: the "green power supply" (GP), used in normal mode, and the "blue power supply" (BP), used in alternate mode.

[107] also describes a preliminary sytem safety analysis of the WBS, using FTA to analyze the "Unannunciated loss of all wheel braking" top event. The resulting FT (Figure 13.2) reflects how the top event depends on the unannounciated loss of the three braking systems and develops the tree downwards, identifying the failures contributing to the unannounciated loss of normal braking. An example of intermediate event is "Normal Brake System does not operate", whereas "Switch failed stuck in intermediate position" is a basic event.

As defined in Chapter 4, we represent a hierarchically organized Fault Tree (FT) [117] as a set of Boolean formulae over Basic Events (BE) and Intermediate Events (IE). This representation defines a tree where leaves are BE, and nodes are IE. For shake of clarity, we report the representation introduced in Definition 4.6.1:

$$FT ::= be \mid ie \mapsto FT \mid FT \wedge FT \mid FT \vee FT$$

According to this definition, the first level of the FT represented in Figure 13.2 can be then expressed as:

"Unannuciated loss of all wheel braking" (the TLE) $\mapsto$
"Loss of all wheel braking" (an Intermediate Event) $\wedge$
"Loss of annunciation capability" (a Basic Event).

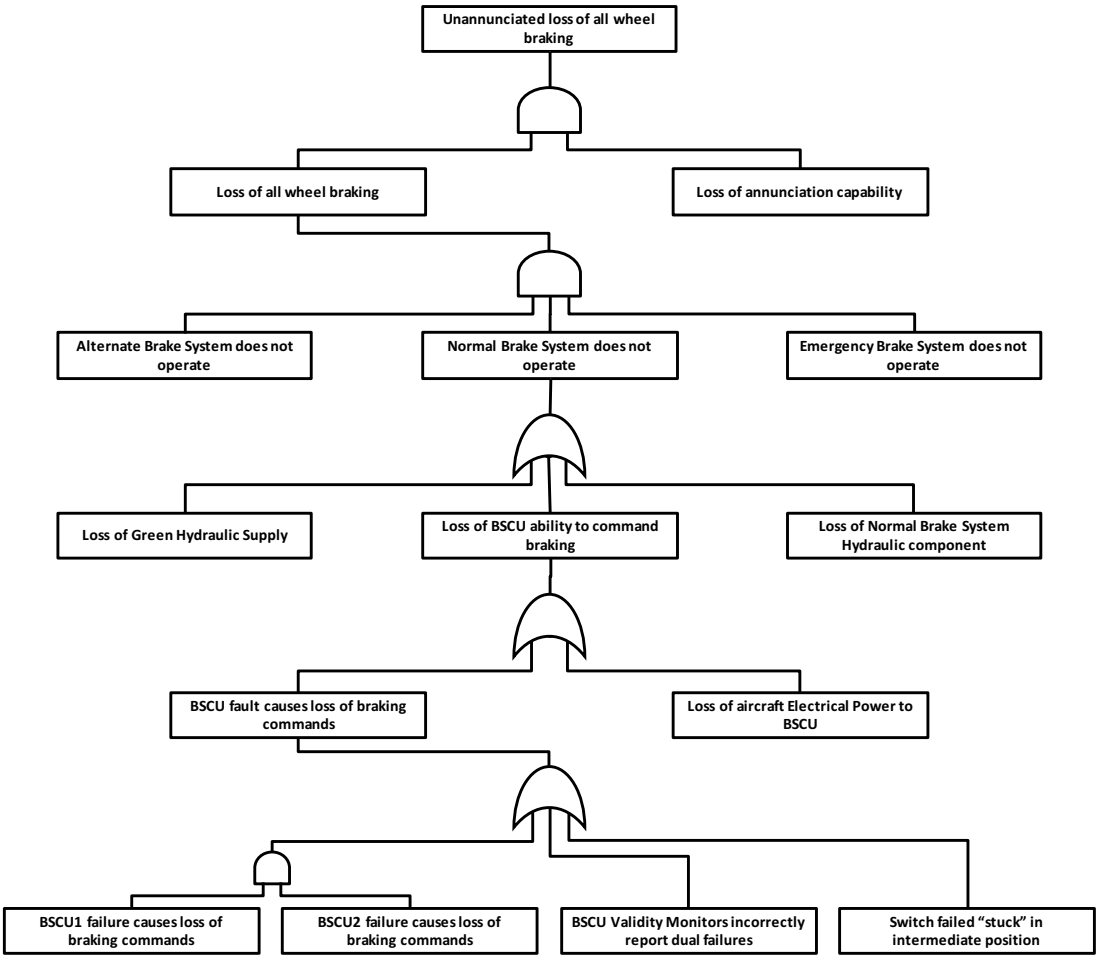


Figure 13.2: Fault tree of an unannounciated loss of all wheel braking developed in [108]

The second level extends the intermediate events of the first one, and in this example it is:

$$\text{“Loss of all wheel braking”} \mapsto$$
$$\text{“Alternate Brake System does not operate”} \wedge$$
$$\text{“Normal Brake System does not operate”} \wedge$$
$$\text{“Emergency Brake System does not operate”}.$$

The successor levels recursively define the IEs, while the Basic Events are treated as terminals, as defined by the representation of a FT.

14

Contract-Based Design

Contract-based design is a paradigm for the design of component based, hierarchically structured, complex systems. This approach has emerged in order to support the fact that the development of such systems is highly distributed, and characterized by a stepwise components refinement. For instance, electrical and hydraulic systems are cooperating actors in an aircraft, but their development is decentralized (sometimes even geographically) and conducted by different engineering teams that share just the interfaces of such systems. Moreover, the development of a (macro) subsystem is carried out by decomposing it in multiple parts.

The contract-based design aims at supporting this process by associating to each component a contract i.e., a clear description of the expected behavior. This definition, expressed over the component's interface, describes a guaranteed behavior over the output ports, provided that the inputs obey a set of assumptions. This Chapter formalizes the concept of Contract-Based Design, which represents the underline theory for the Contract-Based Safety Analysis.

14.1 Components and system architectures

A component interface consists of a set of ports, which are divided into input and output ports[1]. Input ports are those controlled by the environment and fed to the component. The output ports are those controlled by the component and communicated to the environment. Formally, each component S has interface $\langle I_S, O_S \rangle$ of input and output ports. We denote with V_S the set of ports related to the component interface S given by the union of I_S and O_S.

In order to formalize the decomposition, we need to specify the interconnections between the ports, i.e. how the information is propagated around. Intuitively, the input ports of a component are driven by other ports, possibly combined by means of generalized (e.g., arithmetic) gates. These combinations, in the following referred to as *drivers*, depend on the type of the port. Without loss of generality, we assume that ports are either Boolean- or real-valued. The driver for a Boolean port is a Boolean formula; for a real-valued port it is a real arithmetic expression. Therefore, we define a *decomposition* of a component S as a pair $\rho = \langle Sub, \gamma \rangle$ where Sub is a non-empty set of (sub)components such that $S \notin Sub$, and the connection γ is a function that:

- maps each port in O_S into a driver over the ports in $I_S \cup \bigcup_{S' \in Sub} O_{S'}$, and
- for each $U \in Sub$, maps each port in I_U into a driver over the ports in $I_S \cup \bigcup_{S' \in Sub} O_{S'}$.

We extend γ to Boolean and temporal formulas so that $\gamma(\phi)$ is the formula obtained by substituting each symbol s in O_S and I_U for all $U \in Sub$ with $\gamma(s)$. Note that, since ϕ is a Boolean or temporal formula over

[1]For simplicity, we ignore here the distinction between data and event ports.

the ports of a single component, $\gamma(s)$ does not contain s and therefore $\gamma(\phi)$ is well defined (there is no circularity in the substitution).

A system architecture is a tree of components where for each non-leaf component S a decomposition $\langle Sub_S, \gamma_S \rangle$ is defined such that Sub_S are the children of S in the tree. Let Sub^* be the set of components in the architecture tree. Let γ be the union of γ_S with $S \in Sub^*$, i.e., γ takes an expression over $\bigcup_{S \in Sub^*} V_S$ and substitute s with γ_S for every $s \in O_S \cup \bigcup_{S' \in Sub_S} I_{S'}$ (we are assuming that the sets of ports of different components are disjoint). We denote with γ^* the iterative application of γ until reaching a fixpoint. Thus, γ^* takes an expression over $\bigcup_{S \in Sub^*} V_S$ and applies γ until the expression contains only input ports of the root and output ports of the leaf components.

Note that, for simplicity, we are considering only synchronous decompositions for which we need only a mapping of symbols. The framework can be extended to the asynchronous case by considering also further constraints to correlate the ports. In the following, we also assume that we have only one instance for each component so that we can identify the instance with its type to simplify the presentation. In practice, we deal with multiple instances by renaming the ports adding the instance name as prefix.

Example 14.1.1. The WBS architecture, informally introduced in Chapter 13, can be formalized with the notion of decomposition defined above. For example, the top-level system component SC has two subcomponents, namely WBS and BSA. Therefore $Sub(SC) = \{WBS, BSA\}$. The mapping γ is in most of cases just a renaming. For example, the input port P1 of WBS is driven by the input port P1 of SC. Formally $\gamma(WBS.P1) = SC.P1$ (since we avoided the distinction between component types and instances to simplify the notation, we here use the dot notation to have a unique name for each port). In few cases, the driver is not atomic. For example, the

output port Valid of BSCU is driven by the disjunction of the homonyms of SB1 and SB2. Formally, $\gamma(BSCU.Valid) = SB1.Valid \vee SB2.Valid$.

14.2 Trace-Based Components Implementation and Environment

A component S encapsulates a state which is hidden to the environment. It interacts with the environment only through the ports. This interaction is represented by a trace in $Tr(V_S)$.

An input trace is a trace restricted to assignments to the input ports. Similarly, an output trace is a trace restricted to assignments to the output ports. Given an input trace $\sigma^I \in Tr(I_S)$ and an output trace $\sigma^O \in Tr(O_S)$, we denote with $\sigma^I \times \sigma^O$ the trace σ such that for all i, $\sigma[i](x) = \sigma^I[i](x)$ if $x \in I_S$ and $\sigma[i](x) = \sigma^O[i](x)$ if $x \in O_S$.

For simplicity, we do not distinguish between a language (set of traces) and the behavioral model that generates it. Therefore, both implementations and environments of a component S are seen as subsets of $Tr(V_S)$ (note that we are considering also the output ports for the language of the environment because this can be affected by the component implementation).

A decomposition of S generates a composite implementation given by the composition of the implementation of the subcomponents, as well as a composite environment for each subcomponent given by the environment of S and the implementations of the other subcomponents. In order to define formally these notions, we extend γ to states seen as conjunctions of equalities (assignments). Note that, if s is a state, then $\gamma(s)$ represents a set of states. Considering the example of γ introduced in Example 14.1.1, if $BSCU.Valid = \top$, then $\gamma(BSCU.Valid = \top)$ is equal to $(SB1.Valid \vee SB2.Valid) = \top$. Finally, we extend γ to traces seen as sequence of states.

Given a decomposition $\langle Sub, \gamma \rangle$ of S with $Sub = \{S_1, \ldots, S_n\}$ and an implementation M_j for each subcomponent interface $S_j \in Sub$, we define the composite implementation $CI_\gamma(\{M_j\}_{S_j \in Sub(S)})$ of S taking the product of the traces of the subcomponents and projecting on the ports of the component S:

$$\begin{aligned} CI_\gamma(\{M_j\}_{S_j \in Sub(S)}) := \\ \{\pi^I \times \pi^O \in Tr(V_S) \mid \\ \exists \pi_1^O \in Tr(O_{S_1}), \ldots, \pi_n^O \in Tr(O_{S_n}) \text{ s.t.} \\ \pi^I \times \pi_1^O \times \ldots \times \pi_n^O \in \gamma(M_1) \cap \ldots \cap \gamma(M_n) \cap \gamma(\pi^O)\} \end{aligned}$$

Similarly, given a subcomponent $S_h \in Sub$, an implementation M_j for each subcomponent $S_j \in Sub\backslash$ with $j \neq h$, and an environment E for S, we define the composite environment $CE_\gamma(E, \{M_j\}_{S_j \in Sub(S), j \leq h})$ of S_h taking the product of the traces of E and the other subcomponents and projecting on the ports of S_h:

$$\begin{aligned} CE_\gamma(E,\{M_j\}_{S_j \in Sub(S)}) := \\ \{\pi_h^I \times \pi_h^O \in Tr(V_{S_h}) \mid \\ \exists \pi^I \in Tr(I_S), \pi_1^O \in Tr(O_{S_1}), \ldots, \pi_n^O \in Tr(O_{S_n}) \text{ s.t.} \\ \pi^I \times \pi_1^O \times \ldots \times \pi_n^O \in \gamma(M_1) \cap \ldots \cap \gamma(M_n) \cap \gamma(\pi_h^I)\} \end{aligned}$$

14.3 Contracts

A component contract is a pair of properties, called the *assumption*, which must be satisfied by the component environment, and the *guarantee*, which must be satisfied by the component implementation when the assumption holds. We assume as given an assertion language for which every assertion $\mathcal{A}$ has associated a set of variables $V_\mathcal{A}$ and a semantics $L(\mathcal{A})$ as a subset of $Tr(V_\mathcal{A})$. In practice, we will use LTL to specify such assertions, but the approach can be applied to any linear-time temporal logic.

Given a component S, a contract for S is a pair $C = \langle \mathcal{A}, \mathcal{G} \rangle$ of assertions over V_S representing respectively an *assumption* and a *guarantee* for the component. Let M and E be respectively an implementation and an environment of S. We say that M is an implementation satisfying C iff $M \cap L(\mathcal{A}) \subseteq L(\mathcal{G})$. We say that E is an environment satisfying C iff $E \subseteq L(\mathcal{A})$. We denote with $\mathcal{M}(C)$ and with $\mathcal{E}(C)$, respectively, the implementations and the environments satisfying the contract C.

Two contracts C and C' are *equivalent* (denoted with $C \equiv C'$) iff they have the same implementations and environments, i.e., iff $\mathcal{M}(C) = \mathcal{M}(C')$ and $\mathcal{E}(C) = \mathcal{E}(C')$. A contract $C = \langle \mathcal{A}, \mathcal{G} \rangle$ is in normal form iff the complement $\overline{L(\mathcal{A})}$ is contained in $L(\mathcal{G})$. We denote with $nf(C)$ the assertion $\neg\mathcal{A} \vee \mathcal{G}$. The contract $\langle \mathcal{A}, nf(C) \rangle$ is in normal form and is equivalent to (i.e., has the same implementations and environments of) C [19].

Example 14.3.1 (WBS contract)**.** We are interested in defining the contract related to the requirement of the WBS that, given the application of the braking pedals, must activate the brakes. This is formalized with the LTL formula $\mathcal{G} = \mathbf{G}((P1 \vee P2) \rightarrow \mathbf{F}(Brake))$. The WBS component requires an environment that provides the same signal on the pedal application and such that power is always supplied to the BSCU and hydraulic pumps. This is formalized in the LTL formula $\mathcal{A} = \mathbf{G}((P1 = P2) \wedge GP \wedge BP \wedge SP)$.

14.4 Contract refinement

Since the decomposition of a component S into subcomponents induces a composite implementation of S and composite environment for the subcomponents, it is necessary to prove that the decomposition is correct with respect to the contracts. In particular, it is necessary to prove that the composite implementation of S satisfies the guarantee of S's contracts and

that the composite environment of each subcomponent U satisfies the assumptions of U's contracts. We perform this verification compositionally only reasoning with the contracts of the subcomponent independently from the specific implementation of the subcomponents or the specific environment.

In the following, for simplicity, we assume that each component S has only one contract denoted with C_S and is refined by the contracts of all subcomponents (the approach can be easily extended to the general case [58]). Given a component S and a decomposition $\rho = \langle Sub, \gamma \rangle$, the set of contracts $\mathcal{C} = \cup_{S' \in Sub(S)} C_{S'}$ is a refinement of C_S, written $\mathcal{C} \leq_\rho C_S$, iff the following conditions hold:

1. given an implementation $M_{S'}$ for each subcomponent $S' \in Sub(S)$ such that $M_{S'}$ satisfies the contract $C_{S'}$, then $CI_\gamma(\{M_{S'}\}_{S \in Sub(S)})$ satisfies C_S (i.e., the correct implementations of the sub-contracts form a correct implementation of C_S);

2. for every subcomponent S'' of S, given an environment E of S satisfying C_S and an implementation $M_{S'}$ for each subcomponent $S' \in Sub(S)$ such that $M_{S'}$ satisfies the contract $C_{S'}$, then $CE_\gamma(E, \{M_{S'}\}_{S' \in Sub(S)})$ satisfies $C_{S''}$ (i.e., the correct implementation of the other subcomponents and a correct environment of C_S form a correct environment of $C_{S''}$).

Example 14.4.1 (WBS contract refinement)**.** As shown in Fig. 13.1, the WBS component is decomposed into NWBS, AWBS and EWBS. The contracts of these subcomponents are $C_{nwbs} = \langle \mathbf{G}((P1 = P2) \wedge SP \wedge GP), \mathbf{G}((P1 \vee P2) \rightarrow \mathbf{F}(BN)) \rangle$, $C_{awbs} = \langle \mathbf{G}(BP), \mathbf{G}(((P1 \vee P2) \wedge \neg \mathbf{F}(BN)) \rightarrow \mathbf{F}(BA)) \rangle$, $C_{ewbs} = \langle \top, \mathbf{G}(((P1 \vee P2) \wedge \neg \mathbf{F}(BN) \wedge \neg \mathbf{F}(BA)) \rightarrow \mathbf{F}(BE)) \rangle$. The connection are defined in a straightforward way. It is easy to see that the these contracts correctly refine the contract of the WBS

component. We remark that the implementation of the NWBS would be sufficient to ensure the guarantee of the parent component i.e., AWBS and EWBS systems are redundant and play a role only in case of failures.

15

Contract-Based Safety Analysis

15.1 Contract-Based Fault Injection

The goal of our approach is to take as input an architecture enriched with a correct contract refinement and automatically generate a hierarchically organized FT. The idea is to introduce, for each component and for each contract, two failure ports: one representing the failure of the component implementation to satisfy the guarantee, the other representing the failure of the component environment to satisfy the assumption. This step is represented by the arrow labeled 1.1 in Figure 15.1. The connections among such failures are automatically generated and they are later used to produce the FT, as illustrated by label 1.2 in Figure 15.1. The successive refinement of components (i.e., layers 2 and 3 in Figure 15.1) allows us to extend the analysis and generate a more detailed FT. These characteristics of the CBSA approach mimic the recommended practices outlined in [107].

15.1.1 Extension of components and contracts

Given a component interface $\langle I_S, O_S \rangle$ of the component S, we define the extended interface $\langle I_S^X, O_S^X \rangle$ as the interface in which the inputs has been extended with the new Boolean port f_S^I and the output has been extended

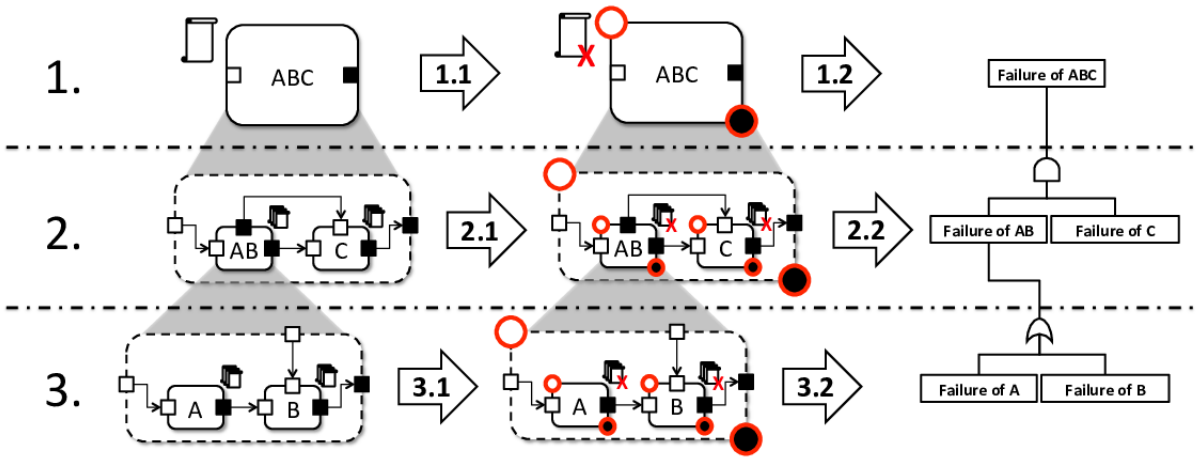


Figure 15.1: Contract-based Safety Assessment Process

with the new Boolean port f_S^O. Namely, $\langle I_S^X, O_S^X \rangle$ is defined as $\langle I_S \cup \{f_S^I\}, O_S \cup \{f_S^O\} \rangle$. Intuitively, f_S^O represents the failure of the component implementation to meet its requirements, while f_S^I represents the failure of the component environment to fulfill the component's assumptions.

The "nominal" contract of a component is extended to weaken both assumption and guarantee, in order to take into account the possible failure of environment and implementation. Given the contract $\langle \mathcal{A}_S, \mathcal{G}_S \rangle$ of S, we define the extended contract $\langle \mathcal{A}_S^X, \mathcal{G}_S^X \rangle$ as follows $\mathcal{A}_S^X = (\neg f_S^I) \rightarrow \mathcal{A}_S$ and $\mathcal{G}_S^X = (\neg f_S^O) \rightarrow \mathcal{G}_S$.

Note that in this simple contract extension the failure is timeless in the sense that either there are no failures and the nominal contract holds, or nothing is assumed or guaranteed. By convention, the failure ports are evaluated initially and the future values are *don't cares*. More complex contract extensions will be developed in the future.

15.1.2 Contract-based synthesis of extended system architecture

We now describe how we generate an extended system architecture given a nominal one with a correct contract refinement. In the extended architecture, components' interfaces and contracts are extended as described in the previous section, while here we automatically synthesize the connections among the extended components. The synthesis ensures that the refinement of contracts in the extended architecture is correct by construction.

For each component S, we define the extended connection mapping γ^X so that $\gamma^X(p) = \gamma(p)$ for all original ports, i.e., for $p \in I_S \cup O_S$, while for the new failure ports γ^X is defined as follows:

- $\gamma^X(f^O_S) := MCS^\top(\gamma((\wedge_{S' \in Sub(S)}(\mathcal{A}^X_{S'} \rightarrow \mathcal{G}^X_{S'})) \wedge \mathcal{A}^X_S), \neg\gamma(\mathcal{G}_S), \{f^I_S\} \cup \{f^O_{S'}\}_{S' \in Sub(S)})$. Intuitively, the driver of the failure of S's guarantee is given by all combinations of the failures of the subcomponents and the environment that are compatible with the violation of the guarantee of S.

- for all $U \in Sub(S)$, $\gamma^X(f^I_U) := MCS^\top(\gamma(\wedge_{S' \in Sub(S)\setminus\{U\}}(\mathcal{A}^X_{S'} \rightarrow \mathcal{G}^X_{S'}) \wedge \mathcal{A}^X_S), \neg\gamma(\mathcal{A}_U), \{f^I_S\} \cup \{f^O_{S'}\}_{S' \in Sub(S)\setminus\{U\}})$. Intuitively, the driver of the failure of U's assumption is given by all combinations of the failures of the other subcomponents and the environment of S that are compatible with the violation of the assumption of U.

The resulting extended contract refinement is correct:

Theorem 15.1.1. *If* $\{C_{S'}\}_{S' \in Sub(S)} \preceq_\gamma C_S$, *then* $\{C^X_{S'}\}_{S' \in Sub(S)} \preceq_{\gamma^X} C^X_S$.

Example 15.1.1 (Synthesis of faults dependencies for WBS component)**.** Given the extended contract C^X_{wbs}, the safety analysis will produce the dependencies formulae for each fault port f^O_{wbs}, f^I_{nwbs}, f^I_{awbs} and f^I_{ewbs}. Specifically, the resulting faults dependency for $f^O_{wbs} := (f^O_{awbs} \wedge f^O_{nwbs} \wedge f^O_{ewbs}) \vee$

($f^I_{wbs} \wedge f^O_{ewbs}$), which means that every assignment of such formula will cause the failure of f^O_{wbs}. This result confirms that the braking ability of the WBS is guarantee if at least one of NWBS, AWBS and EWBS is working, but in case of loss of the power sources (f^I_{wbs}) the EWBS is necessary in order to guarantee the right behaviour. The following analysis for f^I_{nwbs} and f^I_{awbs} will produce respectively $f^I_{nwbs} := f^I_{wbs}$ and $f^I_{awbs} := f^I_{wbs}$. In fact, the subsystems NWBS and AWBS need for BP, SP, and GP power lines, which functionality is part of the assumption of the WBS. The last step addresses the verification of the proof obligation for f^I_{ewbs} which is unsat, expressing the fact that it has no dependencies to the other fault ports. According to this result, Figure 13.1 shows that EWBS is not dependent to any assumptions of the WBS i.e., it does not need any power sources.

15.2 Contract-Based Fault Tree Analysis

15.2.1 Contract-based Fault Tree Generation

Given the extension of the system contract refinement, the FT is automatically generated. The top level event is the failure f^O_S of a non-leaf component S. It is labeled with "Fail of C_S", where C_S is the contract of S. The intermediate events are similarly labeled with the failure of the guarantees of the components that are used in the contract refinement and are not further refined. The failure of the system environment is labeled with "Fail of Environment". The leaves of the tree are basic events, representing the failure of the system's assumption and the failures of the guarantees of contracts that are not further refined. If the architecture is extended further in a step-wise way by decomposing some leaves components, these basic events can become intermediate and be refined further by exploiting the extended contract refinement.

The FT is generated starting from the top level event f^O_S and linking

it to the intermediate events present in $\gamma^X(f_S^O)$. Formally, if f is a basic event, then the FT is atomic: $FT(f) := f$; if f is an intermediate event, then $FT(f_S^O) := f_S^O \mapsto \gamma^X(f_S^O)$. Thus, the FT is defined recursively until reaching the basic events. To simplify the tree, we do not label the failure of the assumption of intermediate components. Therefore, if U is not the system component and f_U^I is present in the tree, we replace it with $\gamma^X(f_U^I)$. Note that the same failure may appear in different branches of the FT – this is standard in FTA – hence, in the above top-down procedure we only need to expand one occurrence of the same failure. We also assume that in the relationship among the failures there is no circular dependency. Usually, such dependencies may be broken by introducing time delays [117]. We leave modeling of faults with temporal dynamics and dealing with circular dependencies to future work.

Example 15.2.1 (Automatic generation of WBS FT)**.** By applying contract refinement to the WBS example, we obtain the FT in Figure 15.2. As it can be seen from Table 15.1, there is nearly a one-to-one mapping with the FT presented in Figure 13.2 – the only differences are that: (i) in the contract-based FT the failure of the environment is considered also for the sub-components that depends to it, and this provides a more detailed system failure explanation; (ii) the monitoring function is more detailed in our model.

15.2.2 CBSA Cut-Sets semantics

We notice that, in the generated FT, the cut sets local to a single component decomposition are minimal by construction. Here, we consider the cut sets of the whole FT that are obtained by replacing intermediate events with their definition in the FT. We call them flattened cut sets, since they can be represented as a two-level FT. They are defined in terms of the

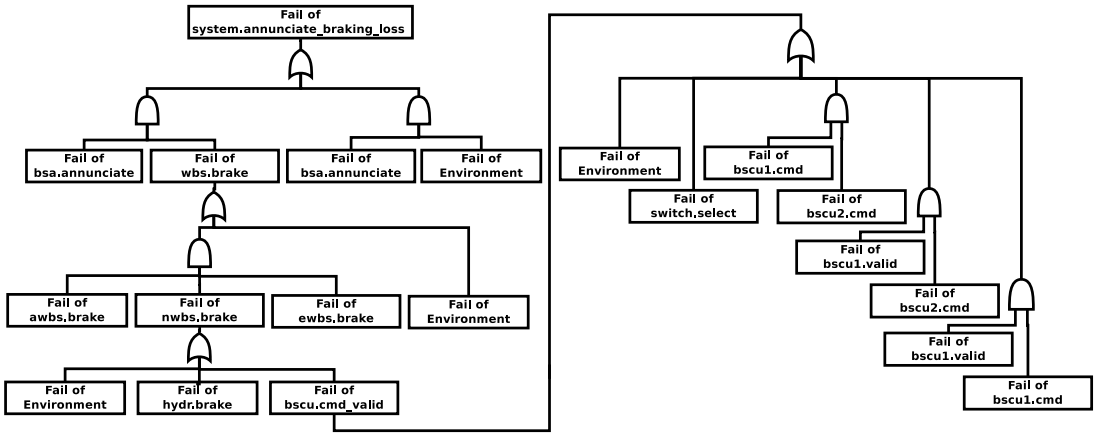


Figure 15.2: Fault tree of an unannunciated loss of all wheel braking: automatically generated

Failure of Contract	**Description**
system.annunciate_braking_loss	Unannunciated loss of All Wheel Braking.
bsa.annunciate	Loss of Annunciation Capability.
wbs.brake	Loss of All Wheel Braking.
nwbs.brake	Normal Brake System does not operate.
awbs.brake	Alternate Brake System does not operate.
ewbs.brake	Emergency Brake System does not operate.
hydr.brake	Loss of Normal Brake System Hydraulic Components.
bscu.cmd_valid	Loss of BSCU Ability to Command Braking.
switch.select	Switch Failure Contributes to Loss of Braking Commands.
bscu1.cmd	Loss of BSCU sub system 1.
bscu1.valid	Loss of monitoring for BSCU sub system 1.
bscu2.cmd	Loss of BSCU sub system 2.
bscu2.valid	Loss of monitoring for BSCU sub system 2.

Table 15.1: Failure of contracts description

failures of the basic components and of the system environment.

Let *leaves* be the basic components of the architecture and let *root* the (root) system component. We denote with $\mathcal{F}$ the set of basic failure ports,

i.e., $\mathcal{F} = \{f_l^O\}_{l \in leaves} \cup \{f_{root}^I\}$, and we identify a fault configuration with an assignment to these parameters. A cut set is therefore a fault configuration of a trace violating the top-level guarantee.

Given a failure port f_S (either input or output) of a component S in the architecture, let us define $\gamma^{X*}(f_S)$ as the iterative application of γ^X to f_S until reaching a fixpoint, i.e., a Boolean combination of failures in $\mathcal{F}$ only. $\gamma^{X*}(f_S)$ defines the set of flattened cut sets obtained with CBSA. We prove that every cut set (in the standard sense) is also a flattened cut set for CBSA.

Theorem 15.2.1. *Let* $L^X = \mathcal{L}(\gamma^*(\wedge_{l \in leaves}(\mathcal{G}_l^X) \wedge \mathcal{A}_{root}^X))$.
If $FC \in CS(L^X, \neg(\mathcal{G}_S), \mathcal{F})$, *then* $FC^\top \models \gamma^{X*}(f_S^O)$.

Here, L^X represents the extension of the system architecture in a MBSA-like fashion, where the guarantees of leaf components and the root assumption are extended locally without explicit constraints among component failures (hence, γ^* is used instead of γ^{X*}). The converse is not true in general. In fact, for the contract refinement to be correct, it is sufficient that the contract of the composite component is weaker than the composition of those of the subcomponents. However, this may create cut sets that are present considering the weaker contract, while are they ruled out by the composition.

15.2.3 Relationship between contracts and generated fault trees

We remark that the FT generated with the proposed approach is clearly sensitive to the contracts and can be used to improve the CBD. For example, in the contract specification of the WBS proposed in [64], each redundant sub-BSCU guarantees that the input pedal application is followed by the braking command or the Validity Monitor set to invalid within a given time bound. Following this approach, the proposed procedure generates a

FT in which each sub-BSCU is a single point of failure. In fact, a failure of its contract means that it can keep the Validity Monitor set to true without ever braking. This contrasts with [107]. The FT shown in Figure 15.2 is actually obtained with an improved specification, where we separated the functional part of the contract from the monitoring of safety, providing a contract that says that every pedal application is followed by the braking command and another contract demanding that the Validity Monitor is set to invalid if the pedal is applied but the brake is never commanded.

16

Experimental Evaluation

We implemented our methodology on top of OCRA [54], a tool for architectural design based on CBD. The OCRA language allows the user to specify contracts (written in various temporal logics of different expressiveness, including LTL and HRELTL [57]), and associate them to architectural components. The correctness of refinements is reduced to a set of proof obligations (as per Chapter 14) – temporal satisfiability checks that are carried out by nuXmv [97], the underlying verification platform, which provides reasoning capabilities via BDD-, SAT-, and SMT-based techniques.

We extended OCRA in the following directions. First, we implemented primitives to automatically extend the architectural description by means of symbolic fault injection, extending the ports and the contracts. Second, we implemented the procedure for the synthesis of the interconnections between failure ports among different levels, as per Section 15.1. Finally, we implemented the procedure to extract FTs from the extended models, as per Section 15.2, and the algorithms used here are based on the ones described in Part II.

We first evaluated the CBSA approach by modeling (several variants of) the WBS case-study in OCRA. The analysis demonstrated very useful to

Faults Number (FN)	9	10	11	12	13	...	29
Minimal Cutsets (MCS)	6	11	22	42	50	...	3316
Mono.BDD	619	1106	3180	T.O.	T.O.	T.O.	T.O.
CBSA.BDD	701	701	701	702	702	702	703
Mono.BMC+BDD	3.1	3.4	4.1	4.6	4.9	...	582
CBSA.BMC+BDD	1.8	1.8	1.8	1.8	1.8	1.8	1.8

Table 16.1: Scalability comparison

provide feedback on the structure of the contracts. In fact, as described in Section 15.2.3, we could improve over the first version of the WBS model described in [64, 58]. We then compared our approach with the "flat" MBSA approach implemented in xSAP - a re-implementation of FSAP[40]. xSAP supports FTA for behavioral models (finite state machines written in the SMV language). We refer to the xSAP approach as *monolithic*, since it generates FTs that are "flat"(i.e., presented as DNF of the MCSs). In OCRA, FTs can be generated from behavioral models, by associating each leaf component with an SMV implementation, where the activation of failure modes causes the violation of contracts. For the evaluation, we associated concrete implementations to the leaf WBS components. We first evaluated the tightness of the contract extension. As described in Section 15.2, CBSA can provide a "pessimistic" interpretation of the system failure, due to the hierarchical partitioning imposed by contract decomposition. Indeed, our results confirm that this is the case for the WBS: if the concrete implementations happen to operate correctly even if the power is not provided, then the monolithic approach provides a tighter set of MCSs. However, if the concrete implementations are such that a loss of power implies a loss of functional behavior, then both techniques result in the same sets of MCSs.

We also compared the scalability of the monolithic and the CBSA ap-

proach for FTA. We considered a parameterized version of the WBS, by varying the total number of faults (NF), and the upper bound for the cycles needed to wait until performing an emergency reaction (DS). The experiments were run on an Intel Xeon E3-1270 at 3.40GHz. We first varied the delay DS (with NF = 9). With NF = 10, CBSA takes 11m40s (BDD), and 2s (BMC+BDD, with k=20), whereas the monolithic approach takes 14m and 7s, respectively. For NF = 15, CBSA times do not vary, while the monolithic approach requires more than 50m (BDD), and 15s (BMC+BDD). The stability in performance shown by the CBSA approach is motivated by the fact that the time needed to compute the FT is mainly spent during the contracts evaluation, whereas analyzing the leaves takes always less than 1s. We then fixed DS = 5 and varied NF from 9 to 29. The results are reported in Table 16.1, where both BDD and BMC+BDD were run with k=20. CBSA is subject only to a marginal degradation in performance, since the variation is local to the computation of the FTs for the leaves. In contrast, the monolithic method passes from 10m19s to timing out after one hour for NF = 12 (BDD), and from 3s to 582s (BMC+BDD). This degradation is directly correlated to the increased number of MCSs, that are enumerated by the monolithic approach. As a final remark, notice that the CBSA approach is fully incremental: the only variation required when exploring different implementations is in constructing the FTs resulting from the analysis of each finite state machine with respect to its contracts. This contrasts with the considerable efforts required in the monolithic approach, that needs to be repeated for each different implementation.

In this particular case study, the BMC+BDD is particularly efficient to compute the Fault Tree (i.e., the CBSA approach takes less than 2 seconds), primarily for the relatively small size of the model. Due to this fact, the applications of the IC3-based approachs described in Part II would not be

appreciable. For a more detailed analysis on the performance of the Fault Tree Analysis we refer to Part VI.

17

Future Extensions

In this part we proposed a new, formal methodology for safety assessment based on contract-based design and automated fault injection. This approach is able to automatically generate hierarchical FTs mimicking system decomposition, and overcoming two key shortcomings of traditional MBSA [41], namely the lack of structure of the generated FTs, and the poor scalability. Moreover, it provides full support to the informal, manual state of the practice, and it can provide important feedback in the early stages of system design.

As future work, we will investigate methods to pinpoint situations where the hierarchical decomposition leads to over-constraining, and to generate suitable diagnostic information. Second, we will generalize fault injection with the introduction of more fine-grained failure dynamics based on temporal patterns and the use of specific fault models (similar to the contract extension with "exceptional" behavior [46]). We will investigate aspects related to fault propagation [2] and extend the framework to consider richer contract specification languages to enable quantitative evaluation of FTs. Moreover, we will extend the approach in order to generate multi-layered FMEA tables.

Part IV

Redundant Architecture Analysis

Introduction

Failure of hardware components is inevitable. Even the most perfect and flawless design should accept this fact. Nevertheless, we cannot avoid to rely on those hardware components, even in highly critical systems such as aircraft, nuclear power plants, and biomedical equipments. In those area of application, a system malfunctioning can cause loss of human lives and environment damage, and this is not an acceptable condition.

An approach that is usually applied consists in integrating the system with additional redundant components that take over in case of failure of the primary ones. Starting from a system with a specific functionality, its extension with redundant techniques requires several additional functionalities such as detecting internal malfunctioning (Fault Detection), identify its cause (Fault Identification), and apply a reconfiguration of the system to solve the problem (Recovery). In modern system design, the phases characterizing the reaction to a component failure are often implemented in a distributed way, where each component implements part of functional, fault detection, fault identification, and fault recovery behavior. In fact, all components may fail, even the monitoring of the failing component e.g., a failure in the reconfiguration part - driven by the monitoring - of the system would be as hazardous as a failure in the main functional part, causing for instance unnecessary actuations.

Few dedicated techniques has been introduced in order to support design and development of a redundant system architecture. In fact, current

approaches are mainly based on manual techniques, and this result in a highly demanding process both in terms of time and cost.

In this Part we introduce a novel technique that allows the system designer to support the design of redundant architectures. This approach integrates in a model-based safety assessment process, and it allows for a preliminary evaluation of the system e.g., when no detailed implementations are yet defined. In fact, the proposed approach imposes that original and redundant systems have to guarantee the same behavior, regardless to which specifically is.

The rest of Part IV is structured as follows:

- **Background**
  - Chapter 18 provides an overview of the common approaches for increasing system reliability;
  - Chapter 19 introduces current analytical techniques;
- **Contributions**
  - Chapter 20 describes an automated techniques based on SMT;
  - Chapter 21 combines the SMT-based technique with predicate abstraction in order to reach a significant performance improvement;
  - Chapter 22 shows the results of the experimental evaluations;
  - Chapter 23 concludes with our future directions.

18

Architectures for Reliability

A key property in high-dependability, safety critical system 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.

The most common technique to achieve high availability for any service is called Clustering. This approach consists in the introduction of a redundancy in software, hardware, or data systems. When a failure occurs, the main system gets substituted by one of the multiple copies present in the same cluster, and this operation is performed in a complete automated manner. Depending on the type of availability that the system has to guarantee, clusters can be configured in any of the following ways:

- *Cold Standby.* In a cold standby approach, if the primary system is operational and not experiencing any issues, then the secondary one is not powered, not operating, and not taking part the service provisioning. In case of a failure in the primary system, the secondary one begins the starting procedure, and when it terminates the two systems are exchanged. The steps that have to be performed during the starting procedure depends on the nature of the system e.g., a data storage might needs to transfer all the data from the primary node.

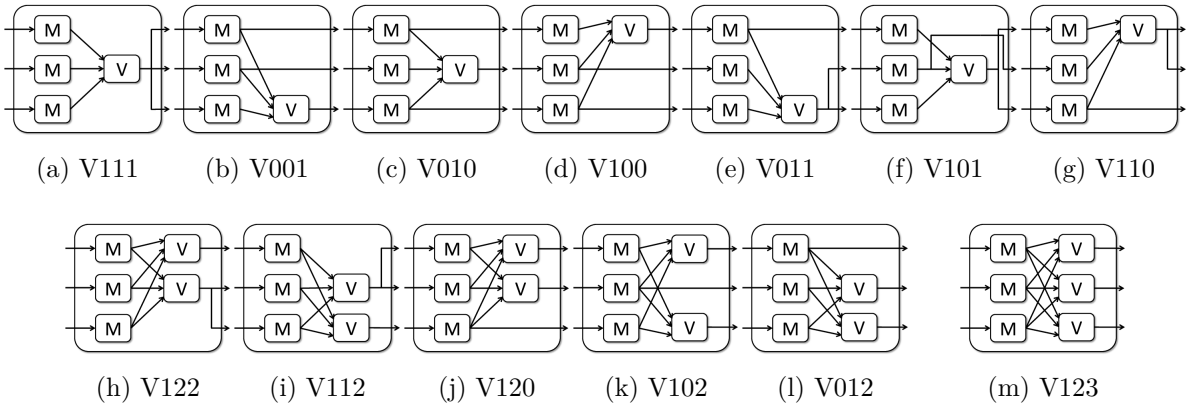
- *Warm Standby.* A redundancy approach is called warm standby when the secondary system is powered, but not operating and not taking part in the service provisioning. The primary and secondary systems are synchronized when a failure occurs in the main one. When this operation terminates the two components are switched. A Warm Standby approach guarantees higher availability than a Cold Standby, because with the first approach the starting procedure has already been performed.

- *Hot Standby.* When relying on a hot standby approach, the secondary system is powered and ready to be operational. When no failures are detected, the secondary system does not process any request or data. The switching mechanism from main to backup system is much faster than both cold and warm standby approaches, since it does not need any additional prearranged configurations.

- *Active-Active*, also called load balanced, is a method where both primary and secondary systems are active and processing requests in parallel. This approach provides, in general, a nearly instantaneous switching between main and backup systems.

In order to increase reliability we consider "active-active" architectures, by means of Triple Modular Redundancy, where multiple copies of the same component are run in parallel, and the results are combined by means of voting to increase the overall likelihood of correct computation. Moreover, "hot-standby" approaches are also considered, where the activation of the backup component is performed by an active component called switch. The approach called Double Modular Redundancy fits into this categorization.

One of the most well-known and widely adopted "active-active" architectural patterns is the Triple Modular Redundancy. This design pattern is widely studied [5, 115, 72, 82, 77, 12] and used for aircraft, nuclear re-



(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

Figure 18.1: Triple Modular Redundancy (1, 2 and 3 voters per stage)

actor plants, railways, and electric supply systems [122, 41, 20, 86]. A basic 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 common values. If only two of the components agree, the value computed by the "majority" is returned. For example, the voter schema used in [122] returns the median of the inputs value. It is easy to see that a TMR schema is able to tolerate a single fault, usually called *single point of failure*.

In order to compose multiple TMR blocks into more complex architectures, triplicated inputs and outputs are considered. TMR blocks may have different numbers of voters (e.g., 1, 2 or 3), and different possible connection between them. The various combinations are shown in Figure 18.1. For the sake of simplicity, we depict unary computing modules within TMR blocks. In fact, computing modules of greater arity are also possible. As an example, consider the architecture from [5], depicted in Figure 19.1, that will be used as a reference example in the rest of this Part. 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_5). On the left-hand side is a description of the data flow being computed, and on

the right a representation of the redundancy architecture.

19

Analysis of Redundant Architectures

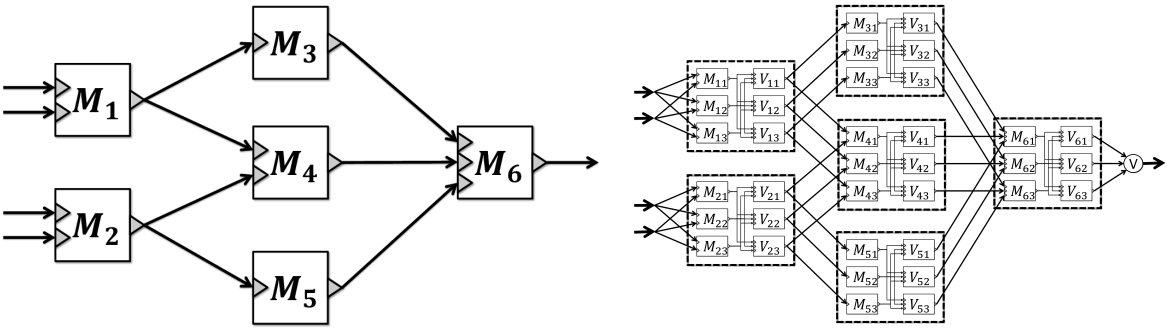
In order to support the design, it is important to provide a set of techniques able to measure the characteristics of a given selection, or even more importantly, the exploration of various architectural choices in the cost-reliability space.

This aspect becomes even more important when considering that the set of possible choices to apply a redundant scheme on a computational network is quite big. In fact, the possible approaches to model the scheme in Figure 19.1, by taking into account the TMRs in Figure 18.1, are in the order of thousands of configurations. The analysis of such redundant schemes becomes even harder when considering that the result should be parameterized on the probability of failure of each component. In fact, having the possibility to rely on different implementations (with higher or lower reliability) might imply the choice of a different redundant scheme.

The following sections describe the techniques that can be used to evaluate an architectural scheme in early design phases.

19.1 State of the practice

Reliability engineering is the discipline that aims at investigating the boundaries between system operation and system failure. In general terms,



Nominal Case Redundant Case

Figure 19.1: Computational Network Example [5]

the techniques that have been developed in this area are mainly oriented on analyzing the reliability performance of a given (redundant) system. Differently, in this Part we concentrate on the evaluation of the impact of applying different redundancy patterns to a given (reference) architecture. Moreover, current approaches lack of a unified model-based approach able to connect the variety of possible analysis [126]. Notable reliability approaches are based on Fuzzy Logics [49], Monte Carlo simulations [127], and Petri Nets [106].

The use of model-based approaches tailored to the analyze different applications of redundant schemes is rather limited. The technique introduced in [72] enable for a semi-automated approach to achieve such kind of evaluations. However, they 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 [82], the formalism of Communicating Sequential Processes (CSP) 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.

In [123], a module based on redundancy is designed within the formalism of timed automata, and analyzed using the Uppaal model checker. This

work focussed on the specific features of the design, and does not consider multi-staged architectures.

An industrial tool developed by Xilinx, called TMRTool [121], supports the automated design of triple modular redundancy for FPGAs systems. However, TMRTool is very specific to FPGAs designes and TMR-based approaches, thus extensions to more generic patterns and architectures are not considered.

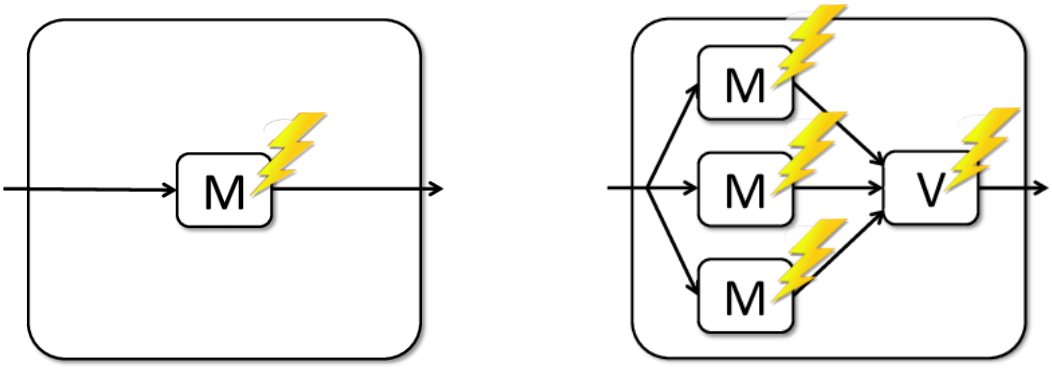
19.2 Comparing Different Redundancy Approaches

One important aspect to consider when dealing with redundant architectures is the probability of failure. In fact, its minimization is the main objective when designing a safety critical and reliable system.

Let us consider the design of a computational component. The basic approach to implement it, as shown in Figure 19.2, considers to describe its behavior with a single module M. In this case, the module M operates as a function f_M, thus the outputs are deterministically computed over the inputs i.e., $f_M(inputs) = outputs$.

If we define the probability P("the module M has a failure") as F_M, then the probability of failure of the computational component is F_M, in fact it depends only to M, and its malfunction represents a *single point of failure*.

A different design of the single module M that relies on a TMR approach, as shown in Figure 19.2, consists in defining 3 copies of M where each of their output is provided as input to the voter V. In this case, the failure of the whole component can be achieved when: (i) 2 modules fail (Expression 19.1), (ii) 3 modules fail (Expression 19.2), or (iii) the voter



Single Module (SM)
$F_{\text{SM}} = F_M$

Triple Modular Redundancy (TMR)
$F_{\text{TMR}} = (1 - F_V) * (3F_M^2 - 2F_M^3) + F_V$

Figure 19.2: Probability of failure example

fails (Expression 19.3).

$$(1 - F_V) * \left(\binom{3}{2} * (1 - F_M) * F_M^2\right) + \tag{19.1}$$

$$(1 - F_V) * \left(F_M^3\right) + \tag{19.2}$$

$$F_V \tag{19.3}$$

The next step consists in evaluating under which conditions a TMR approach is able to increase the system reliability. In fact, a TMR approach is not always able to increase the overall reliability of the system, but this depends on the specific setting where each component implementation is going to operate. More specifically, the reliability analysis of different redundant approach has to take into account the probability of failure of each single module and voter. The 2d plot in Figure 19.3 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

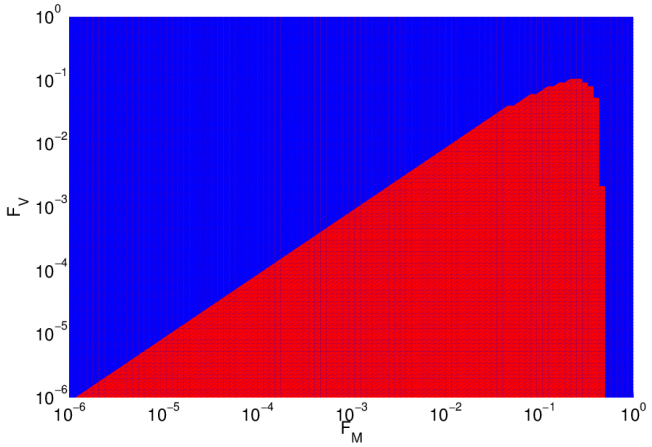
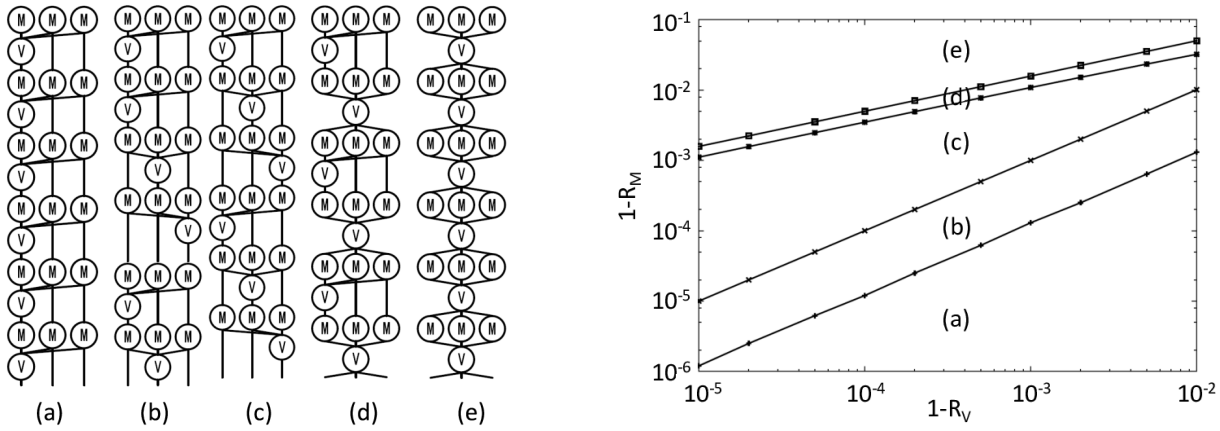


Figure 19.3: Single module (Blue) vs. TMR (Red)

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.

A practical example of the reliability functions evaluations, presented in [72], considers the comparison of different TMR schemes applied to a computational chain of 6 modules, as represented in Figure 19.4. The analysis consists in select the best architectures in the area where $F_V = (10^{-5}, 10^{-2})$ and $F_M = (10^{-6}, 10^{-1})$. The result, shown in Figure 19.4, identifies 5 different areas where each configuration (between (a) and (e)) dominates the others. For instance, if $F_V = 10^{-3}$ and $F_M = 10^{-4}$, then the architecture with the highest reliability is (a), while when $F_M = 10^{-1}$ (e) is the best choice.

An important aspect of the analysis shown in Figure 19.4 is that we are not taking into account the specific behavior of each module M. Moreover, each computational chain is composed of the same number of modules and voters, thus having the same cost (without considering the cable cost). In fact, the result of this evaluation shows how different redundancy patterns may dramatically impact the reliability of the system.



Architectures Considered Best Architectures Evaluation

Figure 19.4: Linear Architectures Comparison.

20

Automated Analysis via SMT

In this Chapter, we propose a novel analysis flow that allows the system designer to assess the reliability of redundant architectures, by means of automated techniques for model-based safety assessment. MBSA provides for a rich modeling framework, where a comprehensive set of architectural solutions is described in an expressive formal logic of equality and uninterpreted functions ($\mathcal{EUF}$). The framework is supported by automated analysis techniques, that allow for the construction of Fault Trees and its relative probabilistic computation. The backend engine, based on model checking, has two key advantages. First, it is based on an expressive modeling language, where it is possible to describe arbitrary redundancy architectures. Second, the flow is fully automated, and allows both to produce fault trees, and to obtain a closed form representation for the reliability function.

20.1 Formal Modeling via SMT

As described in Chapter 18, the application of a redundancy pattern can increase the system reliability i.e., when part of it is not working properly.

The evaluation of a redundant system needs to take into account that aspect, thus the modeling of the coexistence of "good" and "bad" behaviors.

However, the architectural patterns evaluation belongs to the analyses that are tipically performed early in the design phase i.e., when component implementations and behaviors are not yet fully defined.

The SMT-based modeling approach that we defined in this Chapter covers these needs, as it allows for an abstraction of the functional behavior of the system. In particular, it is possible to define nominal and faulty behaviors with two different functions, and integrate both functions in the behavior of the extended components. The nominal function is then shared with all faultless components, in order to guarantee that they have a consistent behavior. Moreover, a faulty component can be described without the addition of any constraint over the faulty function.

Figure 20.1 shows a graphical representation of a TMR with single voter. 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 are sharing the same uninterpreted function (green arrows in Figure 20.1). The faulty part does not implement any specific behavior, thus it 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 20.1 is defined using the SMV language extended with the support for uninterpreted functions. Figure 20.2 presents the definition of the extended module. More in detail, the module receives three parameters: 1. *input*: the input value (of type *Real*); 2. *can_fail*: the parameter that enables the compo-

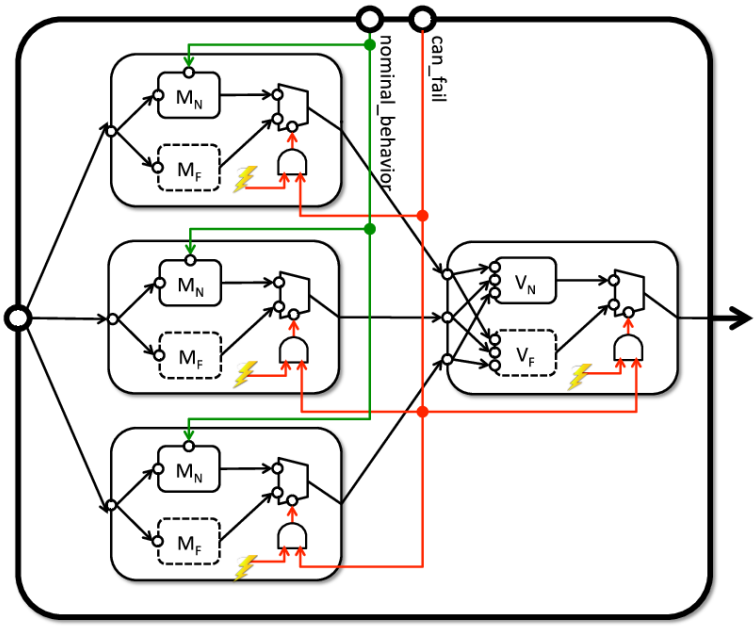


Figure 20.1: TMR component with $\mathcal{EUF}$

nent to have internal failures; 3. *nominal_behavior*: the behavior definition in the nominal case. 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 *faulty_behavior*, and the multiplexer (line *10* in Figure 20.2) that implements the switching between nominal and faulty behavior.

Figure 20.3 presents the definition of the extended voter. More in detail, this component receives five parameters: 1. *input_1*, *input_2* and *input_3*: the input values (of type *Real*); 2. *can_fail*: the parameter that enables the component to have internal failures. In detail, 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 *13* in Figure 20.2) that implements the switching between nominal and faulty behavior. The masking with the *can_fail* signal is represented in line *18*.

```
1 MODULE MODULE_1_INPUT(input, can_fail, nominal_behavior)
2
3 VAR
4   local_failure : boolean;
5
6 FUN
7   faulty_behavior : real -> real;
8
9 DEFINE
10  failure := local_failure & can_fail;
11
12 DEFINE
13  output: =
14    case
15      !failure : nominal_behavior(input);
16      TRUE     : faulty_behavior(input);
17    esac;
```

Figure 20.2: An example of extended module (SMV language)

The modeling approach that we integrated into *SMV* language (with the support for $\mathcal{EUF}$ theory) allows one to model 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 18.1.

20.2 The Miter Composition

The idea at the base of the analysis of redundant architectures is to evaluate which component failure might 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 20.1) gives us the possibility of 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 in case of a faulty description.

```
1 MODULE VOTER_1_TINPUT(input_1, input_2, input_3, can_fail)
2
3 VAR
4   local_failure : boolean;
5
6 FUN
7   faulty_behavior : real * real * real -> real;
8
9 DEFINE
10  voted_output := case
11                    input_1 = input_2 : input_1;
12                    input_1 = input_3 : input_1;
13                    input_2 = input_3 : input_2;
14                    TRUE : input_3;
15                  esac;
16
17 DEFINE
18  failure := local_failure & can_fail;
19
20 DEFINE
21  output: =
22    case
23      !failure  : voted_output;
24      TRUE      : faulty_behavior(input_1, input_2, input_3);
25    esac;
```

Figure 20.3: An example of extended voter module (SMV language)

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 20.4, is called *miter*. 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 the whether reference and faulty outputs are equal. Other possible analysis could be to consider a wrong output only whether it differs to the expected one by a predefined threshold.

A different approach on miter construction is based on a localized staged composition, where the combination of faulty and reference module is called stage. This composition, depicted in Figure 20.5, is similar to a standard miter, considering *can_fail* and *nominal_behavior* instantiation, but in this case faulty and reference inputs are kept separated. The next step is to compose a system architecture, as the one represented in Figure 19.1, and substituting each module with its stage composition such that faulty and

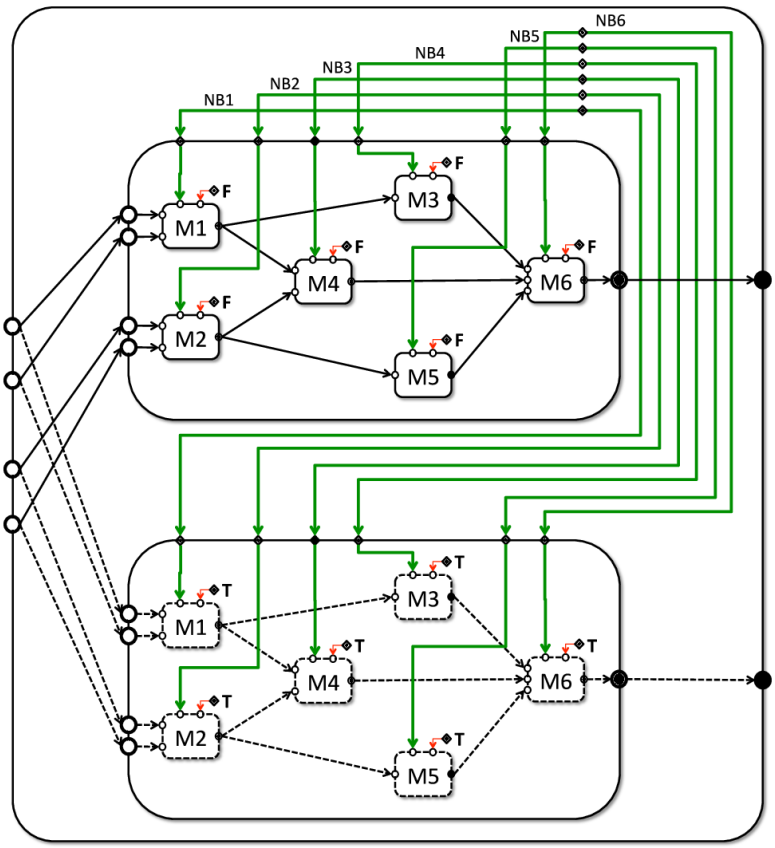


Figure 20.4: Miter composition

reference ports match each other. Such composition is represented in Figure 20.6. The two approaches on miter composition are logically equivalent, in fact they differ only on components displacement. However, the staged miter combines reference and faulty components in the same container, and this characteristic allows for a more refined stage composition. This approach can provide an advantage on the verification performance, considering that each stage can be substituted with simplified implementations as far as they preserve the original behavior. A technique that relies on the staged miter composition to provide significant performance improvement is exemplified in Chapter 21.

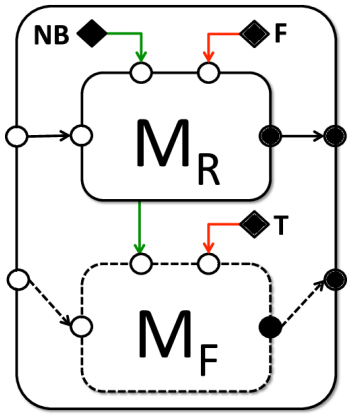


Figure 20.5: Stage composition

20.3 Reliability Evaluation as Fault Tree Analysis

The analysis of a redundant architecture consists in evaluating all possible components failure that may cause the whole system to produce a wrong output. This approach matches one to one with Fault Tree Analysis. Moreover, the fault effects are modeled in order to guarantee that monotonicity assumption does hold. In fact, a fault occurrence enables a completely unconstrained behavior, which means that even the nominal one is allowed. In this way, the monotonicity assumption is satisfied by construction.

Given the miter construction of the TMR in Figure 18.1a, we want to perform a Model-Based Fault Tree Analysis with "two or more outputs of the TMR_faulty are different from the outputs of the TMR_perfect" as Top Level Event. The analysis will give us the Minimal Cutsets $\mathrm{MCS}_{\mathrm{TMR111}} = \{\{\mathrm{M1.fault}, \mathrm{M2.fault}\}, \{\mathrm{M1.fault}, \mathrm{M3.fault}\}, \{\mathrm{M2.fault}, \mathrm{M3.fault}\}, \{\mathrm{V1.fault}\}\}$. The results, represented by the Fault Tree in Figure 20.7, shows that the failure of the voter V1 represents a single point of failure. Such condition, can be though improved by relying on a configuration with triple voter e.g., the pattern in Figure 18.1m. In this case, the MCS

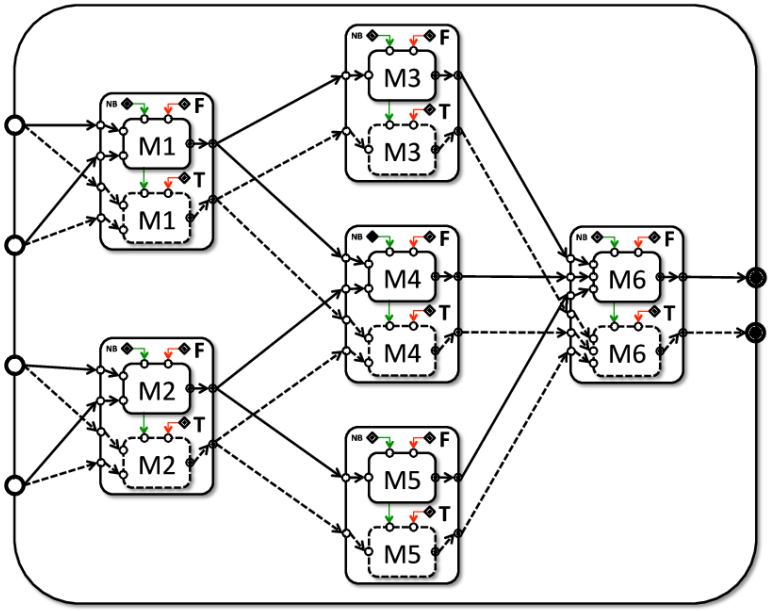


Figure 20.6: Miter composition (stage level)

are $\text{MCS}_{\text{TMR123}}$ = {{M1.failure, M2.failure}, {M1.failure, M3.failure}, {M2.failure, M3.failure}, {V1.failure, V2.failure}, {V1.failure, V3.failure}, {V2.failure, V3.failure}}. The results confirm the increase in system reliability, with the failure of the voter V1 that is no more a single point but now it needs to be combined with either V2 or V3 in order to obtain a wrong output value.

20.3.1 From Fault Tree to Reliability Function

In this section we analyze in detail the techniques used to carry out safety analysis and extract the reliability function that characterizes an architectural description.

Symbolic reliability computation

The algorithm we have described in Section 6.2 for the numerical computation of system reliability can be extended to carry out the support for a

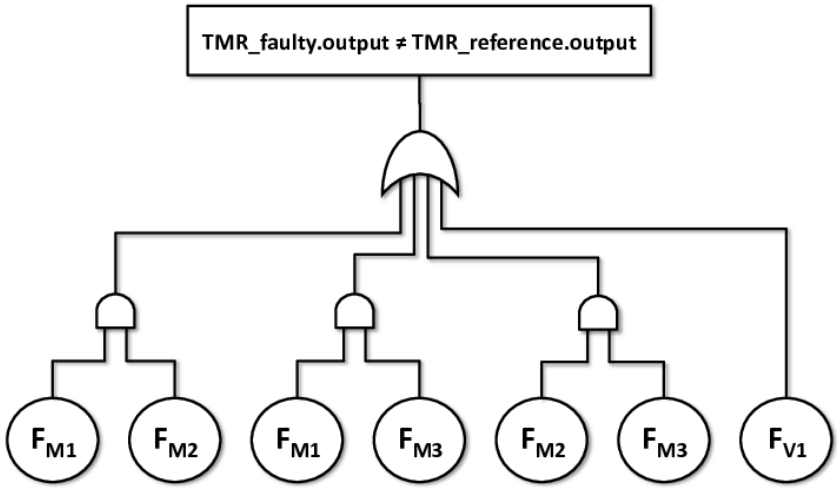


Figure 20.7: Fault Tree for TMR 1V 111 configuration

symbolic evaluation, i.e. automatically 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.

Considering the BDD represented in Figure 20.8, the application of the method in Section 6.2 by relying on symbolic variables instead of numerical values, will provide the equation (20.1). This result represents the reliability function describing the probability of failure of the TMR, assuming that fault events are independent.

$$\begin{aligned} & P(F_{V1}) + \\ & (1 - P(F_{V1}))*(P(F_{M1}) * P(F_{M2}) + \\ & \qquad P(F_{M1}) * (1 - P(F_{M2})) * P(F_{M3}) + \\ & \qquad (1 - P(F_{M1})) * P(F_{M2}) * P(F_{M3})) \end{aligned} \tag{20.1}$$

The reliability function for a TMR-based architecture becomes intractable to compute with manual techniques even with a simple chain

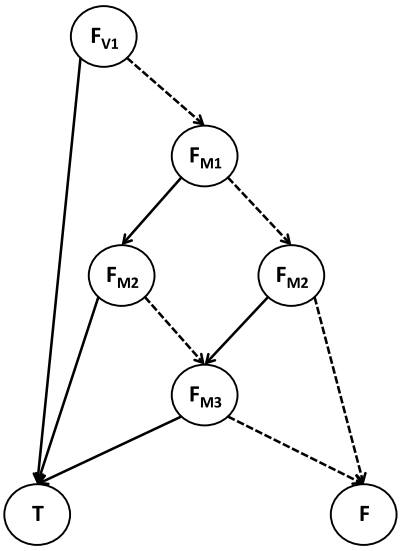


Figure 20.8: BDD representation of the Fault Tree in Figure 20.7

composed of 2 modules. As an example, Equation (20.2) represents the reliability function computed for the configuration V111 (Figure 18.1a) followed by V001(Figure 18.1b). This formula has been obtained automatically by using the symbolic computation techniques of the probability of failures (and then simplified with a symbolic tool in order to make it more compact).

$$\begin{aligned} F_{sys}(F_m, F_v) = & F_v + 2 * F_m * F_v + 6 * F_m^2 - 16 * F_m^4 * F_v^2 + \\ & - 10 * F_v * F_m^2 - 4 * F_m^6 * F_v^2 - 2 * F_m * F_v^2 + \\ & + 4 * F_m^2 * F_v^2 + 4 * F_m^3 * F_v^2 + 14 * F_m^5 * F_v^2 - \\ & + 4 * F_m^3 - 9 * F_m^4 + 25 * F_v * F_m^4 + \\ & + 12 * F_m^5 - 26 * F_v * F_m^5 - 4 * F_m^6 + 8 * F_v * F_m^6 \end{aligned} \tag{20.2}$$

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 architecture. As an example, let us consider three different modules,

M_1, M_2 and M_3, that 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 (i.e. $P_f(M_1)$) is equal to F_{M1}, $P_f(M_2) = 7/8 * F_{M1}$ and $P_f(M_3) = 5/8 * F_{M1}$, can be easily expressed in order to evaluate the overall reliability. Equation (20.3) shows an example of the generated reliability formula, where the failure probability of M_1 is k times the failure of other modules.

$$\begin{aligned} F_{sys}(F_m, F_v, k) = & F_v + 2*F_m*F_v + 2*F_m^2 - 4*F_v*F_m^2 - 4*F_m^4*k^2 - \\ & + 4*F_m^6*k^2 - 2*F_m*F_v^2 - 2*F_m^4*F_v^2 + \\ & + 2*F_m^2*F_v^2 + 2*F_m^3*F_v^2 + 4*k*F_m^2 + 8*F_m^5*k^2 + \\ & - 16*F_v*F_m^5*k^2 - 10*k*F_m^4*F_v^2 - 6*F_v*k*F_m^2 + \\ & - 4*F_m^4*F_v^2*k^2 - 4*F_m^6*F_v^2*k^2 + 2*k*F_m^2*F_v^2 + \\ & + 2*k*F_m^3*F_v^2 + 6*k*F_m^5*F_v^2 + 8*F_v*F_m^4*k^2 + \\ & + 8*F_v*F_m^6*k^2 + 8*F_m^5*F_v^2*k^2 - 4*k*F_m^3 + \\ & - 2*F_v*F_m^3 + 2*F_v*k*F_m^3 - F_m^4 - 4*k*F_m^4 + \\ & + 3*F_v*F_m^4 + 14*F_v*k*F_m^4 + 4*k*F_m^5 + \\ & - 10*F_v*k*F_m^5 \end{aligned} \tag{20.3}$$

20.4 Reliability Functions Evaluation

This Section proposes a set of evaluations on chains of sequential TMR modules with 1 and 2 voters. The idea is to arbitrarily define an array of TMR configurations that represents the pattern that have to be consecutively applied. For each of these patterns we generate the reliability function parameterized by $F_m = (1 - R_m)$ and $F_v = (1 - R_v)$, which represent the failure probability for modules and voters. Moreover, the reliability

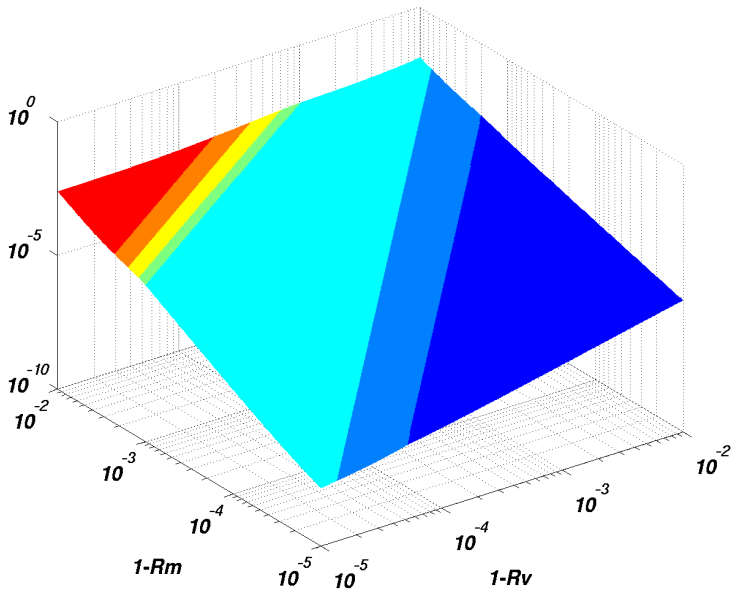


Figure 20.9: 3D view for 1 voter comparison

functions are generated in *Matlab* format and stored together in order to provide a reliability function library of known architectural patterns.

This setting allows us to easily compare the reliability of architectures. For instance, considering the patterns described in Table 20.1, we can compare them together and generate the chart shown in Figure 20.10a. This view highlights, for each pair of values for F_m and F_v, the best configuration. Moreover, this approach allows for the generation of very informative artifacts. In particular, with our approach it is possible to provide a 3-dimensional view of the comparison between chains of TMR with 1 voter. This view is shown in Figure 20.9 and it allows for a clear interpretation of system reliability when varying the probability of failure of each component.

One voter per stage, uniform distribution

The analysis of the TMR with 1 voter consists in evaluating chains of length 8 with patterns of length 4. Moreover, we explicitly added the configura-

tions studied in [72] in order to have a direct comparison with the previous results. The outcome of this analysis is presented in Figure 20.10a where each (colored) area expresses that a specific configuration is better than the others in terms of system reliability. The configurations in Figure 20.10a, explained in Table 20.1, confirm the results presented in previous work, and highlight the power of our approach.

By analyzing the results, we see from Figure 20.10a that the configurations that consider multiple outputs from the voter (e.g. the configurations in Figure 18.1e, 18.1f and 18.1g) are not more reliable than the others, for the considered reliability values.

One voter per stage, non-uniform distribution

As we described in Section 20.3, it is possible to relax the assumption that all modules have the same failure probability. In this way, it is possible to accommodate the trade-off between cost and reliability (module with higher reliability may come at the price of higher cost). In this scenario, we are able to provide the evaluation of redundant systems characterized by non-uniform failure probability for each module. Similarly to the analysis for uniform probability, in Figure 20.10b and Table 20.2, we report the comparison between TMR with $(7/8) * F_m$ for M_1, where M_1 is the left-side module for each configuration in Figure 18.1. The results of this analysis show that, when the module 1 has higher reliability with respect to the others, the best configurations are the ones shown in Figures 18.1a, 18.1d and 18.1c. This result can be explained by the fact that M_2 and M_3 are less reliable than M_1, and in this case the voter is more effective on the modules that have lower reliability.

Two voters per stage, uniform distribution

Similarly to the analysis for 1 voter, we performed a comparison between configurations that consider TMR with 2 voters. The results are reported in Figure 20.10c, with details in Table 20.3. The results of the analysis is similar to the case with 1 voter. In particular, when the reliability of the voter increases the configurations switch gradually from the one in Figure 18.1l (moderate use of voters) to the one in Figure 18.1i (intensive use of voters).

Two voters per stage, non-uniform distribution

The analysis on the reliability of TMR chains with 2 voters and non-uniform probability considers the case when one voter has higher reliability with respect to the other. In detail, we analyze the case of $(1/2) * F_v$ for V_1, where V_1 is the left-side voter for each configuration in Figure 18.1. The higher reliability of the left-side voter imposes the use of configurations that concentrate the computation on this part (left-side) of the TMR, (in particular we are referring to the one shown in Figure 18.1h). When R_v decreases, the best configurations are the ones that minimize the use of voters, and in particular the ones shown in Figures 18.1j, 18.1k and 18.1l.

One voter vs. two voters per stage

An interesting view about the chains of TMR is the comparison between 1 and 2 voters per stage. In particular, we use the standard evaluation in the area of $10^{-5} \sim 10^{-2}$ for x and y axes, as for previous analyses. The results of the evaluation are presented in Figure 20.11a, and they are explained in Table 20.5. In this case, we highlight the difference in the order of magnitude of reliability between the two approaches. In particular, Figure 20.11b shows in red the area where 1 voter is better, and in blue

the area where it is worse. The z axis of this plot represents the value of the difference between two sets of configurations. Analyzing such view, we can see that the approach with 2 voters is clearly better when the reliability of the module is reasonably lower than the reliability of the voter. Differently, when the two reliabilities are comparable, the difference between the approaches is negligible.

System unreliability, proportional evaluation

This analysis evaluates system reliability when varying the ratio between R_v and R_m, with R_v fixed to 10^{-5}. In this work we propose the same evaluation introduced in [87] in order to compare our automated approach with previous results. The configurations with 1, 2 and 3 voters are described in detail in Table 20.6. The results of this analysis are reported in Figure 20.12, where it is shown that the configuration with 3 voters performs better than the others. Moreover, it can be noticed that the standard 1 voter setting is an interesting choice only if the reliability of the voter is not less than $10^2 * R_m$.

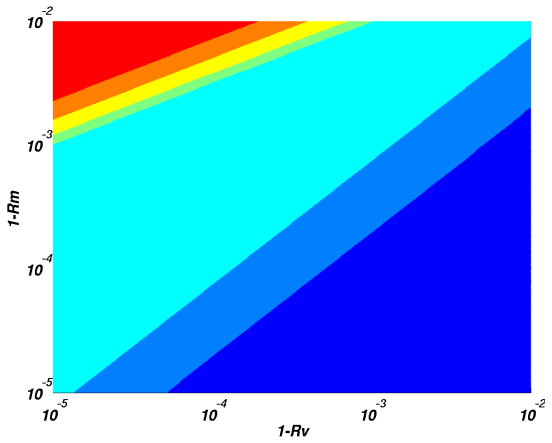
System reliability, varying non-uniform probabilities

The evaluation of system reliability is clearly influenced by the probability distribution of failures that characterize each single component. In view of this fact, we propose an evaluation of system reliability by varying non-uniform distributions for two specific settings. In detail, we analyze the standard TMR chain with 1 voter, described in Table 20.2 configuration (h), and one chain with 2 voters explained by configuration (c) of Table 20.4.

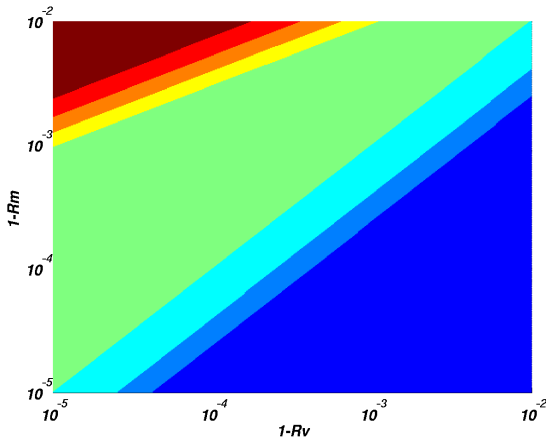
In the first case, we consider the probability of failure for M_1 as $k * F_m$, with k varying from 1/2 to 2. Figure 20.13a shows the results of this

analysis. It is possible to notice that such TMR configurations have an impact on system reliability only when F_m is significantly bigger than F_v. In particular, the probability of system failure is influenced only when $F_m > 10^2 * F_v$.

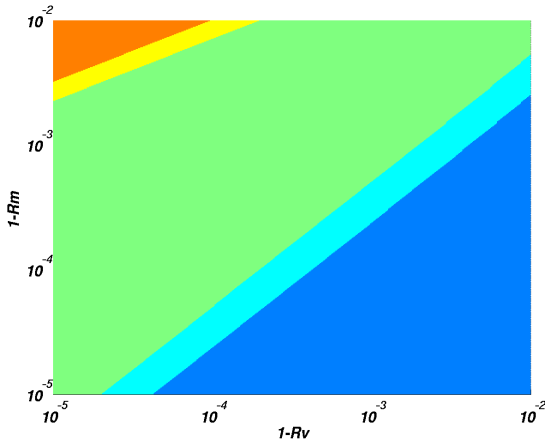
Figure 20.13b shows the evaluation on the configuration with 2 voters. In this case, we consider the probability of failure for V_1 as $k * F_v$, with k varying from 1/4 to 4. Differently from previous analysis, the impact on the reliability of the system is significant only when $F_m < 10 * F_v$. This result can be explained by the fact that, in this area, the reliability of the voter is close to the reliability of the module.



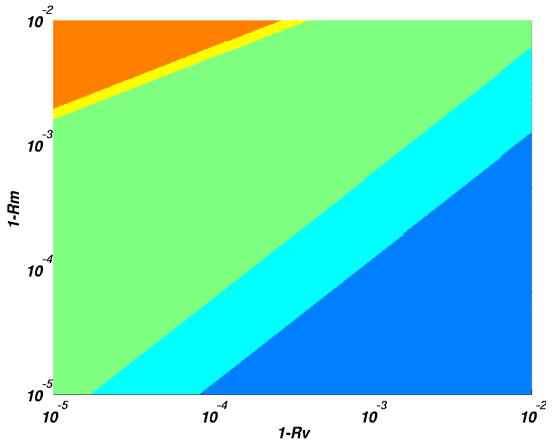
(a) 1 voter: uniform probability



(b) 1 voter: non-uniform probability



(c) 2 voters: uniform probability



(d) 2 voters: non-uniform probability

Figure 20.10: Find best for 1 and 2 voters, uniform and non-uniform probability

Color	Array of configuration
(a) blue	[b,b,b,b,b,b,b,b]
(b) blueberry	[b,b,b,b,c,d,b,c]
(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) orange	[a,a,d,a,a,a,d,a]
(g) red	[a,a,a,a,a,a,a,a]

Table 20.1: Configurations for 1 voters uniform probability

Color	Array of configuration
(a) blue	[d,d,d,d,d,d,d,d]
(b) blueberry	[d,d,d,c,d,d,d,c]
(c) lightblue	[d,c,d,b,d,c,d,b]
(d) green	[d,b,d,c,d,b,d,c]
(e) yellow	[d,c,b,a,d,c,b,a]
(f) orange	[d,a,d,a,d,a,d,a]
(g) red	[a,a,d,a,a,a,d,a]
(h) darkred	[a,a,a,a,a,a,a,a]

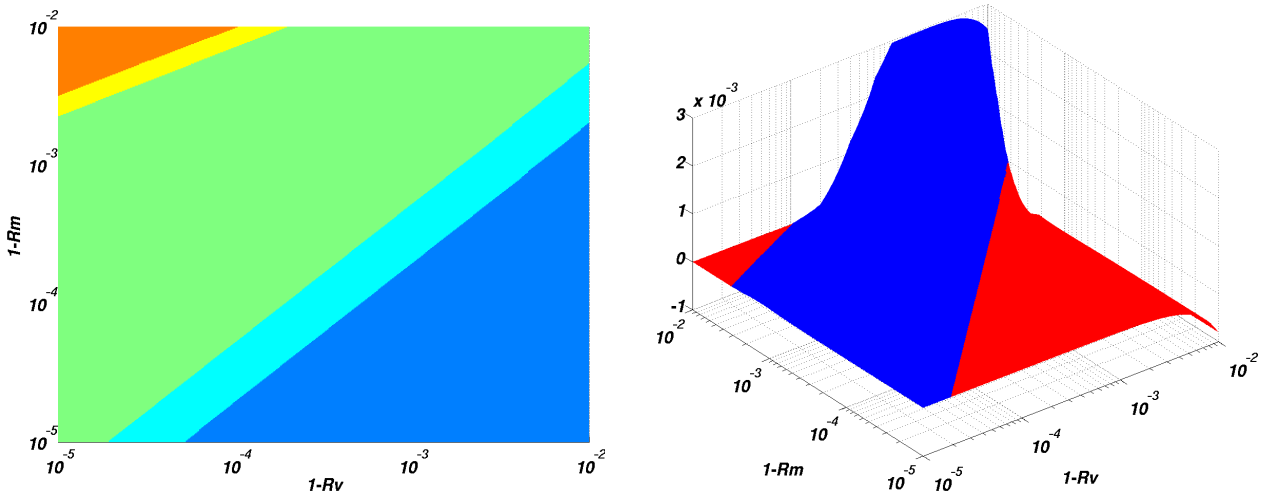
Table 20.2: Configurations for 1 voter non-uniform probability

Color	Array of configuration
(a) blue	[l,l,l,l,l,l,l,l]
(b) lightblue	[l,l,k,j,l,l,k,j]
(c) green	[l,k,j,k,l,k,j,k]
(d) yellow	[i,i,i,l,i,i,i,l]
(e) orange	[i,i,i,i,i,i,i,i]

Table 20.3: Configurations for 2 voters uniform probability

Color	Array of configuration
(a) blue	[l,l,l,l,l,l,l,l]
(b) lightblue	[k,k,j,k,k,j,k,k]
(c) green	[j,k,l,j,k,l,j,k]
(d) yellow	[h,l,h,h,l,h,h,l]
(e) orange	[h,h,h,h,h,h,h,h]

Table 20.4: Configurations for 2 voter non-uniform probability



(a) 1 and 2 voters comparison

(b) 1 voter vs 2 voters (blue means 2v is better)

Figure 20.11: 1 voter vs 2 voters

Color	Array of configuration
(a) blue	[b,b,b,b,b,b,b,b] (1v)
(b) lightblue	[b,b,b,b,c,d,b,c] (1v)
(c) green	[l,k,j,k,l,k,j,k] (2v)
(d) yellow	[i,i,i,l,i,i,i,l] (2v)
(e) orange	[a,a,a,a,a,a,a,a] (1v)

Table 20.5: Configurations for 1 voter vs. 2 voters

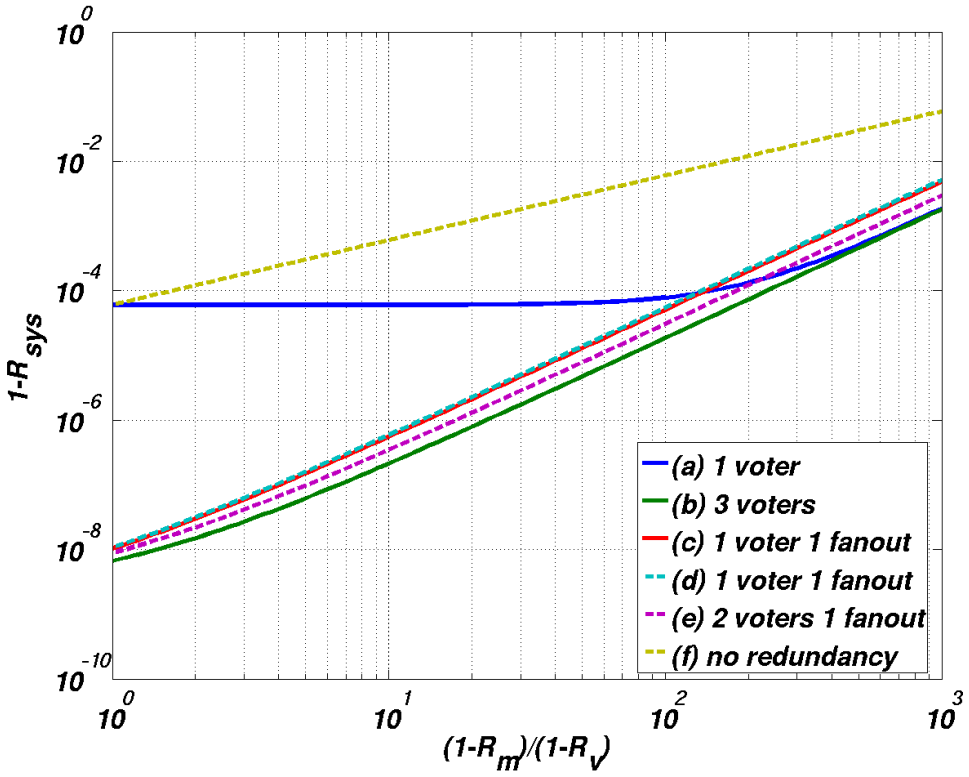
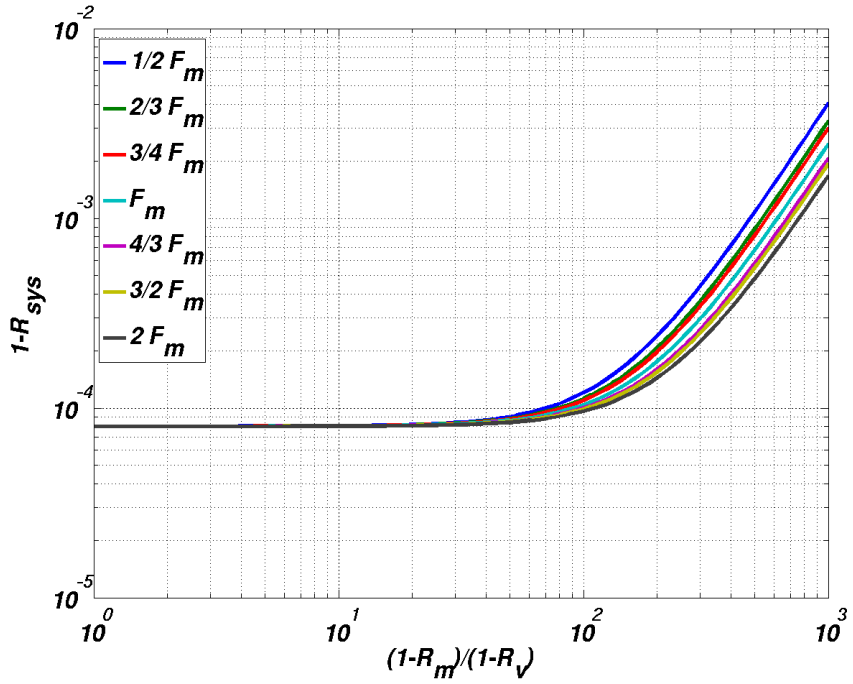


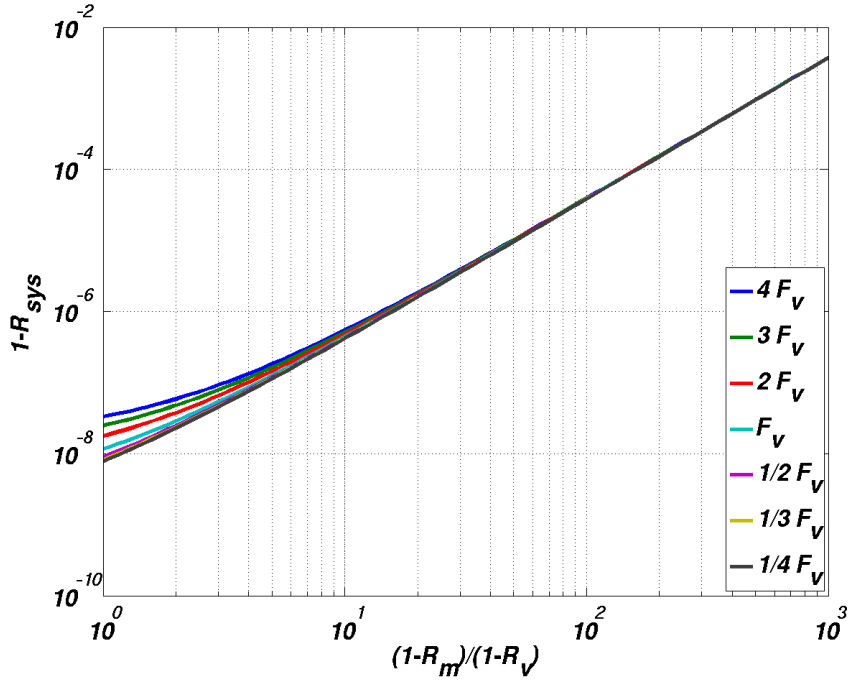
Figure 20.12: System reliability: proportional evaluation

Identification	Description	Array of configurations
(a)	standard 1 voter	[a,a,a,a,a,a] (1v)
(b)	standard 3 voters	[m,m,m,m,m,m] (3v)
(c)	1 voter with 1 fanout	[b,c,d,b,c,d] (1v)
(d)	1 voter with 1 fanout	[b,c,d,d,c,b] (1v)
(e)	2 voters with 1 fanout	[k,l,j,k,l,j] (2v)
(f)	no redundancy	

Table 20.6: Configurations for system reliability: proportional evaluation



(a) Varying F_m for M_1



(b) Varying F_v for V_1

Figure 20.13: System reliability when varying non-uniform probability

21

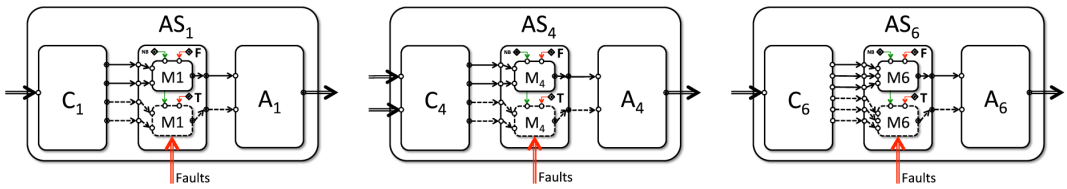
CutSets computation via Predicate Abstraction

A problem of the flow described in previous Chapter is the use of parameter synthesis over SMT to construct the set of CSs, and in a combinatorial system this can be reduced to an All-SMT problem that generates a DNF of the result. Thus, its performance is related to the number of cut sets, and in realistic cases this enumeration can be highly expensive. We now present a method to overcome this bottleneck. The idea is to rely on a suitable predicate abstraction, so that a unique, SMT-based quantifier elimination can be transformed in a BDD-based quantification on a Boolean formula.

We first notice that the miter construction relies on identical copies of the architecture: the architecture under analysis, and the reference architecture, constrained not to fail. There is a clear structural correspondence: every fallible module in the architecture under analysis has a corresponding infallible module in the reference architecture.

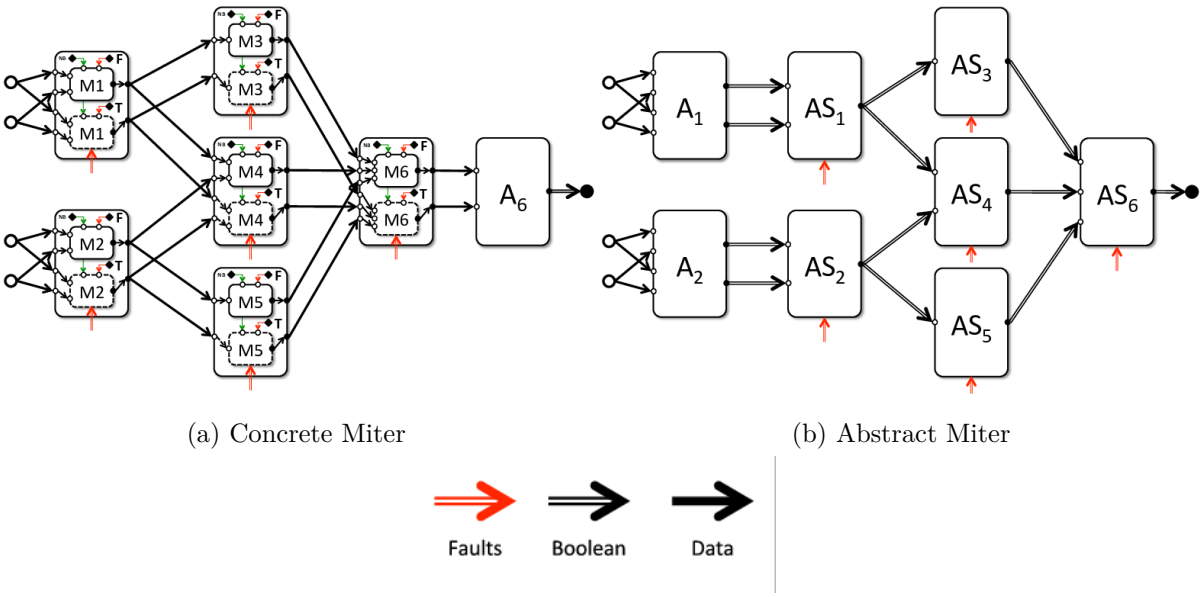
The optimized method for CS construction exploits this structural correspondence by applying a predicate abstraction on input and output ports of each stage, while mirroring the fault variables i.e., they are already in the Boolean domain.

This abstraction can be intuitively represented by connecting one addi-



(a) AS_1 (b) AS_4 (c) AS_6

Figure 21.1: Abstract Stages Example



(a) Concrete Miter (b) Abstract Miter

Figure 21.2: Miter Approaches

tional component to input and output ports of each stage, as represented in Figure 21.1. The component that preprocesses the input signal, called "Concretizer" (modules C_1, C_4, and C_6), receives the predicates as input and provides as output an instance of the concrete signals that satisfies such predicates. Analogously, the output signals are provided to an "Abstractor" (modules A_1, A_4, and A_6) that gives as output the assignment to the predicates.

Considering the miter composition at the stage level described in Chapter 20, and linking its outputs to an abstractor, the resulting system, shown

in Figure 21.2a, is called *concrete miter*. Replacing then each stage with its abstract counterpart (as in Figure 21.1), we obtain a pure Boolean model. The successive addition of an abstractor that preprocesses the inputs allows us to obtain an architecture that has the same interface as the *concrete* one. This model, depicted in Figure 21.2b, is called *abstract miter*.

The significance of the *abstract* model results from the fact that, under some preconditions, it has the very same cut sets as the *concrete* one. In fact, in this chapter we prove an equivalence theorem that supports the overall correctness. The theorem requires specific conditions on the sets of predicates used in the abstraction, together with the proof that the selected predicates satisfy these conditions for all the modules used in the architecture.

21.1 Formal Characterization

We now introduce a formal notation to describe architectures, which allows for the definition of redundant and reference systems, or their miter composition.

The base block of the formalism describing a redundant system is the *Basic Combinatorial Component*. Defined in 21.1.1, it models a system with input and output ports, a set of faults signals and an SMT 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 21.1.1 (Basic Combinatorial Component)**.** A basic combinatorial component is a tuple $\langle \vec{P_I}, \vec{P_O}, F, \pi \rangle$, where:

- $\vec{P_I}$ and $\vec{P_O}$ are the terms representing respectively input and output ports, and each of them can have Boolean ($\mathbb{B}$) or Data ($\mathbb{D}$) type. $\tau(\vec{P}_{I/O}[i])$ represents the type of the i-th input/output port;

- F is the set of faults events of Boolean type;
- $\pi(\vec{P_I}, \vec{P_O}, F)$ is an SMT formula over ports and faults, where each term belongs to $\mathbb{B}$ or $\mathbb{D}$.

More complex architectures are obtained by combining Basic Components into *Combinatorial Components*, as stated in Definition 21.1.2. This notation uses two composition operators: *sequential composition* ($\triangleright$), and *parallel composition* ($|$). The former relates components that are connected in a sequential fashion, linking outputs of the first one with inputs of the former. Parallel composition, on the other hand, juxtaposes the set of ports from different components, which run in parallel.

Definition 21.1.2 (Combinatorial Component)**.** A combinatorial component CC is defined either by:

- Basic Combinatorial Component (Definition 21.1.1);
- $CC \triangleright CC$ as sequential composition (Definition 21.1.4);
- $CC \mid CC$ as parallel composition (Definition 21.1.5).

The sequential composition can only by applied when two components are compatible. This notion is formally defined in 21.1.3, and it essentially prescribes that two components can be connected if the have the same inputs/outputs cardinality, and if their ports are one-by-one of the same type.

Definition 21.1.3 (Sequential compatibility)**.** Given two combinatorial components $M' = \langle \vec{P'_I}, \vec{P'_O}, F', \pi' \rangle$ and $M'' = \langle \vec{P''_I}, \vec{P''_O}, F'', \pi'' \rangle$, they are sequentially compatible, denoted $M' \rightsquigarrow M''$, iff $|\vec{P'_O}| = |\vec{P''_I}|$ and $\forall_{i \in \{0..|\vec{P'_O}|\}} \tau(P'_O[i]) = \tau(P''_I[i])$.

Definition 21.1.4 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''.

Definition 21.1.4 (Sequential composition semantics)**.** Given two combinatorial components $M' = \langle \vec{P'_I}, \vec{P'_O}, F', \pi' \rangle$ and $M'' = \langle \vec{P''_I}, \vec{P''_O}, F'', \pi'' \rangle$, such that $M' \rightsquigarrow M''$, the sequential composition $M = M' \triangleright M''$, where $M = \langle \vec{P}, F, \pi \rangle$, is defined as:

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

Similarly to the sequential case, the parallel composition of two components is defined in 21.1.5, however in this case no compatibility conditions are necessary.

Definition 21.1.5 (Parallel composition semantics)**.** Given two combinatorial components $M' = \langle \vec{P'_I}, \vec{P'_O}, F', \pi' \rangle$ and $M'' = \langle \vec{P''_I}, \vec{P''_O}, F'', \pi'' \rangle$, such that $F' \cap F'' = \emptyset$, the parallel composition $M = M' | M''$, where $M = \langle \vec{P}, F, \pi \rangle$, is defined as:

- $\vec{P_I} = \vec{P'_I} \mid \vec{P''_I}$;
- $\vec{P_O} = \vec{P'_O} \mid \vec{P''_O}$;
- $F = F' \cup F''$;
- $\pi(\vec{P_I}, \vec{P_O}, F) = \pi'(\vec{P'_I}, \vec{P'_O}, F') \wedge \pi''(\vec{P''_I}, \vec{P''_O}, F'')$.

This framework enables the definition of any tree- or DAG-shaped structure. However, relying solely on parallel and sequential compositions might seems to be limiting in order to reach this level of expressiveness. However,

some special basic components can used to connect inputs and outputs ports in a different fashion than just parallel and sequential compositions. In fact, we identified three components to cover the following purposes: duplication of values (module D), simple propagation of input values (I module, a.k.a. identity) and arbitrary reconfiguration of signals (R module). The application of this modeling approach is shown in the Equation 21.1, which represents the system in Figure 19.1. In this case, we use D modules in order to duplicate outputs of the M_1 and M_2 components.

$$(M_1|M_2) \triangleright (D|D) \triangleright (M_3|M_4|M_5) \triangleright M_6 \tag{21.1}$$

Giving the intuition of how a Triple Modular redundancy can be expressed with the formalism introduced here, Example 21.1.1 describes the details of a possible encoding of a single module triplication, as in Figure 20.1.

Example 21.1.1 (TMR as a Combinatorial Component)**.** Giving a single computational module with 1 input and 1 output, which behaviour is defined by the uninterpreted function $B_M : \mathbb{D} \to \mathbb{D}$, then its Triple Modular Redundancy with 3 voters can be represented as a combinatorial component $T = \langle \vec{P_I}, \vec{P_O}, F, \pi \rangle$ such that:

- $\vec{P_I} = [i_1, i_2, i_3]$;
- $\vec{P_O} = [o_1, o_2, o_3]$;
- $F = \{f_{M1}, f_{M2}, f_{M3}, f_{V1}, f_{V2}, f_{V3}\}$;
- $\pi(\vec{P_I}, \vec{P_O}, F) = \exists B_M : \mathbb{D} \to \mathbb{D}, \wedge_{i=1..3} o_i = TMR(\vec{P_I}, B_M, F, f_{Vi})$.

and where the formula TMR is defined as:

$$\begin{aligned}
TMR(\vec{P_I}, B_M, F, f_V) &:= V(M(i_1, B_M, f_{M1}), M(i_2, B_M, f_{M2}), \\
&\quad M(i_3, B_M, f_{M3}), f_V) \\
M(i, B_M, f_M) &:= \neg f_M \rightarrow B_M(i) \\
V(i_1, i_2, i_3, f_V) &:= \neg f_V \rightarrow B_V(i_1, i_2, i_3) \\
B_V(i_1, i_2, i_3) &:= \text{IF } i_1 = i_2 \text{ THEN } i_1 \text{ ELSE } e_1 \\
e_1 &:= \text{IF } i_1 = i_3 \text{ THEN } i_1 \text{ ELSE } e_2 \\
e_2 &:= \text{IF } i_2 = i_3 \text{ THEN } i_2 \text{ ELSE } i_3
\end{aligned}$$

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

As described earlier in this Chapter, of particular interest is to generate a stage composition out of a combinatorial component. This transformation is a special case of the parallel composition, and it is formally defined in 21.1.6.

Definition 21.1.6 (Stage composition)**.** Given a combinatorial component $M = \langle \vec{P_I}, \vec{P_O}, F, \pi \rangle$ its stage composition $\mathcal{S}^M = \langle \vec{P_I''}, \vec{P_O''}, F'', \pi'' \rangle$, considering $M' = \langle \vec{P_I'}, \vec{P_O'}, F', \pi' \rangle$ as a copy of M, is defined as:

- $\vec{P_I''} = \vec{P_I} \mid \vec{P_I'}$;
- $\vec{P_O''} = \vec{P_O} \mid \vec{P_O'}$;
- $F'' = F$;
- $\pi''(\vec{P_I''}, \vec{P_O''}, F'') = \exists F' : \pi(\vec{P_I}, \vec{P_O}, F) \wedge \pi'(\vec{P_I'}, \vec{P_O'}, F') \wedge_{f' \in F'} f' = \bot$.

21.1.1 Systems equivalence

We now define some algebraic properties of combinatorial components. Two combinatorial components are equivalent, as stated in Definition 21.1.7, if their relational formulas have the same value for each assignment to input and output ports, and faults.

Definition 21.1.7 (System equivalence)**.** Given two combinatorial components $M' = \langle \vec{P'_I}, \vec{P'_O}, F', \pi' \rangle$ and $M'' = \langle \vec{P''_I}, \vec{P''_O}, F'', \pi'' \rangle$, such that $F' = F''$, $\vec{P'_I} = \vec{P''_I}$, and $\vec{P'_O} = \vec{P''_O}$, they are called system equivalent, denoted $M' \equiv M''$, if and only if

$$\forall M = \langle p_{I1}, \ldots, p_{In}, p_{O1}, \ldots, p_{Om}, f_1, \ldots, f_{Ik} \rangle : \pi'(M) \Longleftrightarrow \pi''(M).$$

Important result that allow us to manipulate the combinatorial components in order to prove the soundness of our approach is represented by the Lemmas of reduction (21.1.1), parallel (21.1.2), and invertion (21.1.3) equivalence. In particular, Lemma 21.1.1 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 21.1.2 states a similar result for parallel composition.

Lemma 21.1.1 (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 21.1.2 (Parallel equivalence)**.** *Given the combinatorial components S'_1, S''_1, S'_2, S''_2, if $S'_1 \equiv S''_1$ and $S'_2 \equiv S''_2$ then $S'_1 | S'_2 \equiv S''_1 | S''_2$.*

The invertion equivalence Lemma, defined in 21.1.3, shows that the application of sequential and parallel composition can be inverted when the sequential compatibility allows for the former composition to be applied.

Lemma 21.1.3 (Invertion equivalence)**.** *Given the combinatorial components S'_1, S''_1, S'_2, S''_2, if $S'_1 \rightsquigarrow S''_1$ and $S'_2 \rightsquigarrow S''_2$ then $S'_1 | S'_2 \triangleright S''_1 | S''_2 \equiv S'_1 \triangleright S''_1 | S'_2 \triangleright S''_2$.*

Abstraction and Concretization

We now define two special types of components, whose purpose is to formalize the abstraction. Specifically, the *abstractor component* (Definition 21.1.8) is used to translate a set of concrete (data) values into their abstract counterpart, whereas the *concretizer component* (Definition 21.1.9) generates instances of concrete values satisfying the predicates.

Definition 21.1.8 (Abstractor)**.** A combinatorial component $A = \langle \vec{P_I}, \vec{P_O}, F, \alpha \rangle$ is an abstractor iff:

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

Definition 21.1.9 (Concretizer)**.** A combinatorial component $C = \langle \vec{P_I}, \vec{P_O}, F, \gamma \rangle$ is a concretizer iff:

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

By way of abstractors and concretizers, we can express the abstraction of a component M as the sequential composition $C \triangleright M \triangleright A$, and in the Example 21.1.2 we show how to apply the abstraction to a stage composition.

Example 21.1.2 (Stage abstraction)**.** As described in Chapter 20, a miter based approach relies on the comparison between faulty and reference (i.e., faultless) models, and this concept is also applied to the predicates used to abstract stage compositions. For instance, if we consider the combinatorial component T as in the Example 21.1.1, and its stage composition $\mathcal{S}^T$, then the abstract stage will be $C \triangleright \mathcal{S}^T \triangleright A$ where:

- $\gamma([pi_1, pi_2, ..., pi_{2n}], [po_1, po_2, ..., po_n], \emptyset) = \wedge_{i=1..n}\, po_i \Longleftrightarrow pi_i = pi_{(n)+i}$
- $\alpha([pi_1, pi_2, ..., pi_m], [po_1, po_2, ..., po_{2m}], \emptyset) = \wedge_{i=1..m}\, pi_i \Longleftrightarrow po_i = po_{(m)+i}$

The stage composition, as in Definition 21.1.6, represents a specialization of a parallel composition between a module M and its copy M', where the fault variables of the second one are imposed to be false. Intuitively, if M has m output ports, then $\mathcal{S}^M$ has $2m$ of them, where the ports from 1 to m are the outputs from M and the ones from $m+1$ to $2m$ are the reference values computed by the faultless component M'.

$\diamond$

Now we now can formally define concrete and abstract miters, and in the next Section we show how these two systems produce the same result out of the minimal cutsets computation.

21.2 Proof of correctness

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 21.2.1, which 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 21.2.1 (Modular abstraction equivalence)**.** *For all $i \in \{1..n\}$, for all $j \in \{1..m_i\}$ let $S_{i,j}$ be combinatorial components and let $C_{i,j}$ be concretizers. For all $i \in \{1..(n+1)\}$, for all $j \in \{1..m_i\}$ be $A_{i,j}$ abstractors. Let $C_{i,j} \rightsquigarrow S_{i,j}$ and $S_{i,j} \rightsquigarrow A_{i,j}$.*

Let

$$\begin{aligned} \mathcal{C}(S) &= L_n \triangleright \ldots \triangleright L_2 \triangleright L_1 \triangleright A_1 \\ \mathcal{A}(S) &= A_{n+1} \triangleright CLA_n \triangleright \ldots \triangleright CLA_2 \triangleright CLA_1 \end{aligned}$$

where

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

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

$$A_{i+1,j} \triangleright C_{i,j} \triangleright S_{i,j} \triangleright A_{i,j} \quad \equiv \quad S_{i,j} \triangleright A_{i,j} \tag{21.2}$$

then

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

Proof. **Parallel case**

First, we need to prove that if Equation 21.2 holds then that property can be lifted to layers. For all $i \in \{1, \ldots, n\}$

$$L_i \triangleright A_i \equiv A_{i+1} \triangleright C_i \triangleright L_i \triangleright A_i \tag{21.3}$$

Base case

When $m_i = 1$, we have $S_1 \triangleright A_1 \equiv A_2 \triangleright C_1 \triangleright S_1 \triangleright A_1$, that follows from the hypothesis of the theorem.

Step case

Assuming that the property of Equation 21.3 holds for n, then if $S_{i,n+1} \triangleright A_{i,n+1} \equiv A_{i+1,n+1} \triangleright C_{i,n+1} \triangleright S_{i,n+1} \triangleright A_{i,n+1}$ for Lemma 21.1.2 we obtain

$$\frac{\left(\begin{array}{c} S_{i,1} \\ \hline S_{i,2} \\ \hline \cdots \\ \hline S_{i,n} \end{array}\right) \triangleright \left(\begin{array}{c} A_{i,1} \\ \hline A_{i,2} \\ \hline \cdots \\ \hline A_{i,n} \end{array}\right)}{S_{i,n+1} \triangleright A_{i,n+1}} \equiv \frac{\left(\begin{array}{c} A_{i+1,1} \\ \hline A_{i+1,2} \\ \hline \cdots \\ \hline A_{i+1,n} \end{array}\right) \triangleright \left(\begin{array}{c} C_{i,1} \\ \hline C_{i,2} \\ \hline \cdots \\ \hline C_{i,n} \end{array}\right) \triangleright \left(\begin{array}{c} S_{i,1} \\ \hline S_{i,2} \\ \hline \cdots \\ \hline S_{i,n} \end{array}\right) \triangleright \left(\begin{array}{c} A_{i,1} \\ \hline A_{i,2} \\ \hline \cdots \\ \hline A_{i,n} \end{array}\right)}{A_{i+1,n+1} \triangleright C_{i,n+1} \triangleright S_{i,n+1} \triangleright A_{i,n+1}} \tag{21.4}$$

then, for Lemma 21.1.3, Equation 21.5 holds.

$$\frac{\left(\begin{array}{c} S_{i,1} \\ \hline S_{i,2} \\ \hline \cdots \\ \hline S_{i,n} \end{array}\right)}{S_{i,n+1}} \triangleright \frac{\left(\begin{array}{c} A_{i,1} \\ \hline A_{i,2} \\ \hline \cdots \\ \hline A_{i,n} \end{array}\right)}{A_{i,n+1}} \equiv \frac{\left(\begin{array}{c} A_{i+1,1} \\ \hline A_{i+1,2} \\ \hline \cdots \\ \hline A_{i+1,n} \end{array}\right)}{A_{i+1,n+1}} \triangleright \frac{\left(\begin{array}{c} C_{i,1} \\ \hline C_{i,2} \\ \hline \cdots \\ \hline C_{i,n} \end{array}\right)}{C_{i,n+1}} \triangleright \frac{\left(\begin{array}{c} S_{i,1} \\ \hline S_{i,2} \\ \hline \cdots \\ \hline S_{i,n} \end{array}\right)}{S_{i,n+1}} \triangleright \frac{\left(\begin{array}{c} A_{i,1} \\ \hline A_{i,2} \\ \hline \cdots \\ \hline A_{i,n} \end{array}\right)}{A_{i,n+1}} \tag{21.5}$$

Sequential case

We now prove, for all n, that

$$\begin{array}{ccccccccccccc} & & L_n & \triangleright & L_{n-1} & \triangleright & \ldots & \triangleright & L_2 & \triangleright & L_1 \triangleright A_1 & \equiv & \\ A_{n+1} & \triangleright & (C_n \triangleright L_n \triangleright A_n) & \triangleright & (C_{n-1} \triangleright L_{n-1} \triangleright A_{n-1}) & \triangleright & \ldots & \triangleright & (C_2 \triangleright L_2 \triangleright A_2) & \triangleright & (C_1 \triangleright L_1 \triangleright A_1) & & \end{array} \tag{21.6}$$

Base case

When $n = 1$, we have $L_1 \triangleright A_1 \equiv A_2 \triangleright C_1 \triangleright L_1 \triangleright A_1$, that follows directly from Equation 21.3.

Step case

Assume that the property of Equation 21.6 holds for n, by Lemma 21.1.1 it is possible to prepend L_{n+1}.

$$\begin{aligned} L_{n+1} \triangleright L_n \triangleright \ldots \triangleright L_2 \triangleright L_1 \triangleright A_1 \equiv \\ L_{n+1} \triangleright A_{n+1} \triangleright C_n \triangleright L_n \triangleright A_n \triangleright C_{n-1} \triangleright \ldots \triangleright L_1 \triangleright A_1 \end{aligned} \tag{21.7}$$

by Lemma 21.1.1, it is possible to prepend L_{n+1}:

$$\begin{aligned} \underline{L_{n+1}} \triangleright L_n \triangleright \ldots \triangleright L_2 \triangleright L_1 \triangleright A_1 \equiv \\ \underline{L_{n+1}} \triangleright A_{n+1} \triangleright C_n \triangleright L_n \triangleright A_n \triangleright C_{n-1} \triangleright \ldots \triangleright L_1 \triangleright A_1 \end{aligned} \tag{21.8}$$

then if $L_{n+1} \triangleright A_{n+1} \equiv A_{n+2} \triangleright C_{n+1} \triangleright L_{n+1} \triangleright A_{n+1}$ for Lemma 21.1.1 we obtain

$$\begin{aligned} L_{n+1} \triangleright A_{n+1} \triangleright C_n \triangleright L_n \triangleright A_n \triangleright C_{n-1} \triangleright \ldots \triangleright L_1 \triangleright A_1 \equiv \\ A_{n+2} \triangleright C_{n+1} \triangleright L_{n+1} \triangleright A_{n+1} \triangleright C_n \triangleright L_n \triangleright A_n \triangleright C_{n-1} \triangleright \ldots \triangleright L_1 \triangleright A_1 \end{aligned} \tag{21.9}$$

Equation 21.9 proves that the property of 21.6 holds also for $n+1$, as shown in Equation 21.10.

$$\begin{array}{ccccccccc} & & L_{n+1} & \triangleright & L_n & \triangleright & \ldots \triangleright & L_1 \triangleright A_1 & \equiv \\ A_{n+2} & \triangleright & C_{n+1} \triangleright L_{n+1} \triangleright A_{n+1} & \triangleright & C_n \triangleright L_n \triangleright A_n & \triangleright & \ldots \triangleright & C_1 \triangleright L_1 \triangleright A_1 & \end{array} \tag{21.10}$$

□

The results stated in Theorem 21.2.1 is very general; it can be applied to different abstractions, provided that the hypothesis of the theorem holds. Most importantly, the proved equivalence in the formulas of abstract and concrete systems guarantees also to obtain the same result for the MCS computation. In fact, this computation is deterministic given a formal model and a top level event, which in this case is expressed over the output ports.

22

Experimental Evaluation

In this Chapter we compare the two approaches described in this Part, in particular the direct SMT verification against the BDD-based relying on predicate abstraction technique.

22.1 The instantiation

We instantiate the framework described above using a specific abstraction that expresses, given a set of input and output ports, the equivalence between each of the signals and the reference value. More precisely, let us considering a stage with a reference component having i_n, o_n as input and output ports, and a redundant module duplicating the signals with $i_1, i_2, i_3, o_1, o_2, o_3$ as ports, our abstraction generates the predicates $\{(i_n = i_1), (i_n = i_2), (i_n = i_3)\}$ as input, and $\{(o_n = o_1), (o_n = o_2), (o_n = o_3)\}$ as output.

In order to use the results of Chapter 21, we must prove that the hypothesis of Theorem 21.2.1 holds for the selected predicates on all the TMR modules implementing the library. For this purpose, we relied on MathSAT5 [56] to prove that for each staged module, represented as π_γ where π_α is its abstracted counterpart, the following property holds:

$$\text{For all } I \in \mathbb{R} \text{ and } O, F \in \mathbb{B}$$
$$\pi_\alpha(\alpha(I), O, F) \Longleftrightarrow \pi_\gamma(I, \gamma(O), F)$$

The generation of Fault Trees, in the form of Binary Decision Diagrams [47], 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 21.1, 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 18.1, 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 described in Chapter 20, 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 Part II; 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.

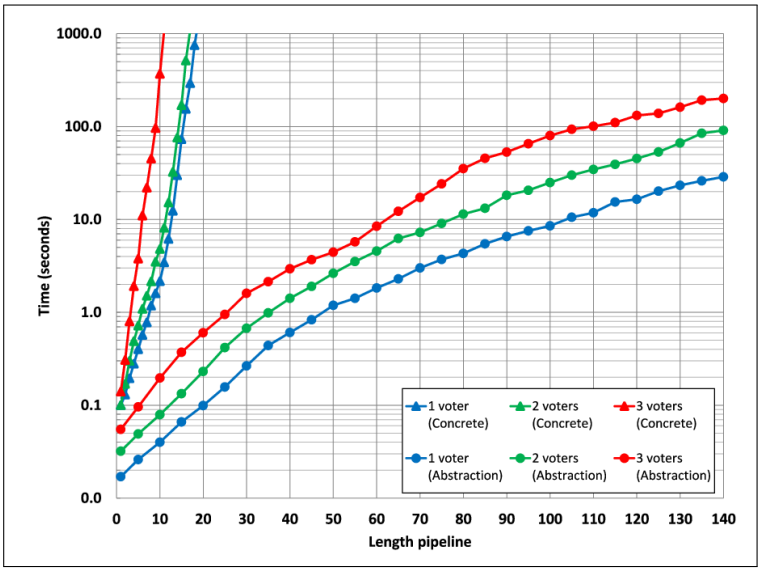


Figure 22.1: Scalability evaluation on linear structures

22.2 Scalability Analysis

We compared the performance of the monolithic and compositional approaches on a set of benchmarks that randomly generates linear-, tree- and DAG- like 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.

22.2.1 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 22.1: the x axis represents the length of the chain, while on the y axis there is the time needed to compute the MCS. 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

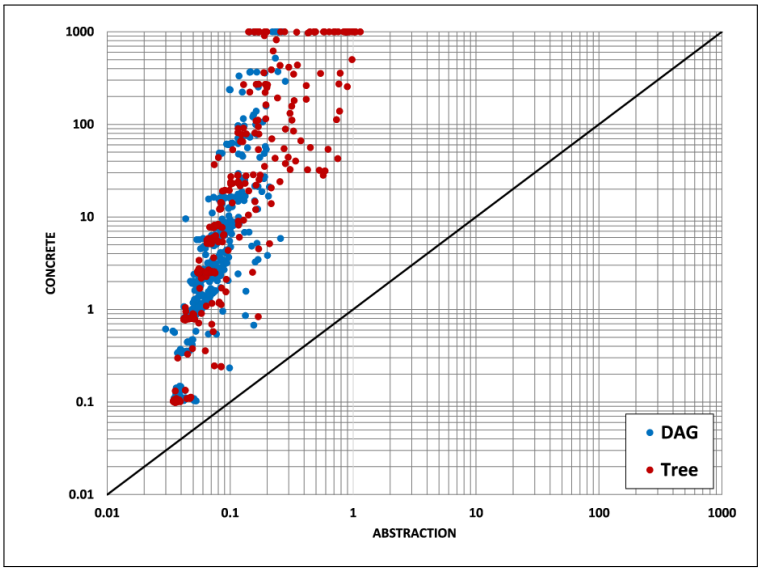


Figure 22.2: Tree (Red) and DAG (Blue) comparison

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 Cutsets. 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.

22.2.2 Scalability on Tree and DAG structures

We then analyzed tree and DAG diagrams, first considering the design description presented in [5], that describes a DAG redundant structure as shown in Figure 19.1. 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 MCS.

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.

The random generation of Tree and DAG networks allows us to compare the performances of two approaches. Figure 22.2 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 22.3. 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 MCS. 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

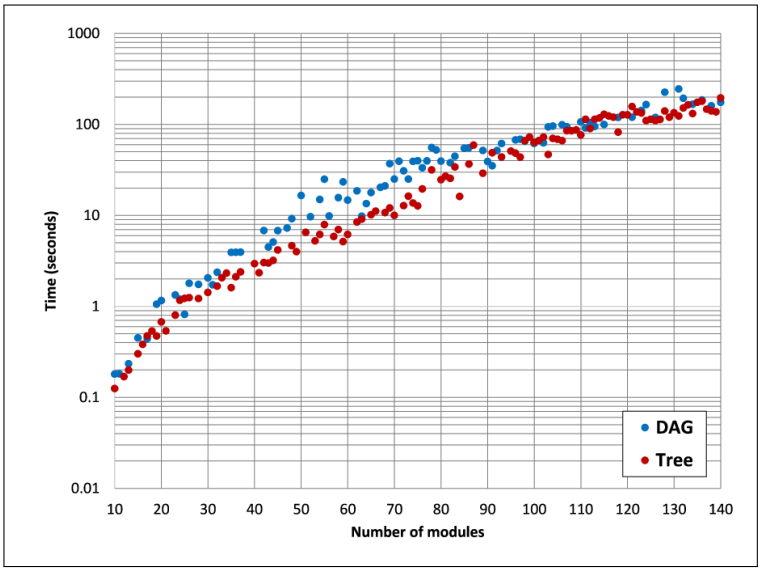


Figure 22.3: Tree and DAG scalability: abstraction

quantification of predicates.

In the monolithic case, the bottleneck is clearly the AllSMT procedure (with optimizations described in [37]), 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.

23

Future Extensions

The main target for future extensions of this work is to allow for automated synthesis of redundant architectures. The idea consists in providing a reference architecture, a set of constraints describing the leaf components expected behavior, and an objective function. Out of these problem description, the methodology should provide a set of redundant architectures that are *Pareto optimal* according with the objective and reliability functions. In fact, a high level of reliability requires redundant components, but each of them adds weight, and cost, to the entire system. Thus, the best architecture is the one that reaches higher safety and reliability while not exceeding weight and cost constraints.

Additional analyses will consider also optimal components displacement. In fact, physical damage of a part of the system (e.g., a localized impact of an object) may affect the functionality of the components that are installed in the damaged area. If those components are part of the same redundant system, then the physical damage would represent a single point of failure e.g., common cause failure.

The techniques described in this Part can be also applied to the Reliability Block Diagram (RBD) analysis. RBD is an inductive method used to analyze system reliability, by representing it with parallel and sequen-

tial sub-blocks. The analysis performed over RBD are directed to discover the root causes of system failures by inspecting the relations between system blocks. Further extensions of our techniques would either analyze a given RBD, by adding the possibility to express abstract relations between components, or producing the resulting RBD out of a given architecture, by generating an abstract system that expresses only relations between components.

Part V

Tools and Integrated Processes

Introduction

Part II, III, and IV describe a set of techniques that aim at optimizing different phases of the formal analysis of a safety critical system. This target can be achieved in practice only by developing the tools that implement those techniques, and include them into a well defined formal process. More specifically, in this work we extended a pre-existent tools architecture, which provides symbolic model checking and contract-based design capabilities. The resulting platform allowed us to apply these new features to a set of real-world case studies.

In this Part we describe the verification toolset that represents the foundations of this work, and how we have designed their extension in order to define a comprehensive formal validation, verification, and safety assessment platform.

The rest of Part V is structured as follows:

- Chapter 24 introduces the verification platforms that represent the bases of this work;
- Chapter 25 describe the implementation choices for Minimal Cutsets Computation algorithms, Contract-Based Safety Assessment, and Redundant Architecture Analysis;
- Chapter 26 defines the formal process that covers all techniques discussed in this Thesis;

- Chapter 27 concludes with the final remarks of this Part.

24

Verification Platforms

The techniques developed in this thesis build upon a complex tool architecture that has been developed since the introduction of the first version of the NuSMV model checker in 1999. Improvements and extensions went primarily through BDD and BMC based model checking, followed by the integration with the MathSAT SMT solver that opened up to infinite states verification. This resulted into the definition of nuXmv, a model checker for finite and infinite states systems with the support for more recent verification techniques like IC3/PDR [43].

nuXmv, being a state of the art model checker [25], it is used as a verification engine for a variety of formal verification tools. Most notably, xSAP [27] provides model-based safety assessment such as Fault Tree Analysis and Fault Mode and Effect Analysis (FMEA) tables generation via minimal cutsets computation. OCRA [54] implements a contract-based paradigm that relies on temporal logic for the components contract definition. Figure 24.1 shows the relation between the aforementioned tools, and in this Chapter we provide an overview of those verification tools.

Figure 24.1: Pre-existent Software Architecture

24.1 NuSMV

In 1999, the NuSMV model checker [53] was developed from a joint project between Carnegie Mellon University (CMU) and FBK (formally, Istituto per la Ricerca Scientifica e Tecnologica (IRST)). NuSMV is a BDD based model checker that redesigns SMV [92] (Symbolic Model Verifier) with a new software architecture and the support for LTL verification. The successive NuSMV2 [51] open source project, in 2002 added the support for SAT based invariant and LTL model checking. NuSMV2 implements standard falsification [24], induction [65], and interpolation [93] based model checking.

24.2 MathSAT

The first version of MathSAT [14] integrated a DPLL-based SAT solver with a decision procedure for linear arithmetic logic. The newer version 5 of MathSAT [56] supports a rich set of theories such as equality and uninterpreted functions ($\mathcal{EUF}$), arrays ($\mathcal{AR}$), linear arithmetic over rational ($\mathcal{LA}(\mathbb{Q})$), integer ($\mathcal{LA}(\mathbb{Z})$), mixed rational-integer ($\mathcal{LA}(\mathbb{QZ})$), fixed width

bit-vectors ($\mathcal{BV}$), and floating-point arithmetic ($\mathcal{FP}$).

24.3 nuXmv

The nuXmv symbolic model checker [50] is the evolution of NuSMV, which has been extended to support infinite states systems and adding the integration with modern verification techniques such as IC3/PDR [43]. The nuXmv model checker supports also LTL model checking with the integration of k-liveness [60] and IC3.

The support for infinite states systems of nuXmv is available via the integration with an SMT solver, which in this case is MathSAT5. The SMV language has been also extended, in order to support infinite domains variables such as Integers and Reals.

24.4 xSAP

The Formal Safety Analysis Platform (FSAP) [40] is a toolset that aim at supporting design and safety engineers in the development of a complex and safety critical system. FSAP relies on NuSMV as backend engine, and extends it in order to provide BDD based algorithms for the minimal cutsets computation [37]. FSAP provides also a fault injection based model extension for SMV models.

xSAP [27] extends FSAP with a new set of algorithms and analyses. The fault injection now supports the definition of more complex behaviors like multiple failures and common cause. The model checking engines are based on nuXmv, which provides the support for SAT-based algorithms for finite and infinite states systems. In addition, xSAP provides fault propagation analysis based on Timed Failure Propagation Graphs (TFPG), and Common Cause Analysis (CCA).

24.5 OCRA

OCRA [54] is a tool for the verification of logic-based contract refinement. It implements the assume-guarantee paradigm, and it supports specification and analysis of component-based system architectures. Each component is defined with an interface of input and output ports, where their expected behavior is expressed via discrete or hybrid linear-time temporal logic. This approach allows for a compositional modeling and verification by relying on the refinement structure of the system. nuXmv is used by OCRA as a backend engine to deal with the temporal logic satisfiability problem.

25

Tools Implementation

25.1 Minimal Cutsets Computation

The architecture of nuXmv is composed of several parts, called add-ons, each of them devoted to solving a specific problem. An add-on extends the core system functionality, and are specialized to provide a set of verification techniques such as BMC, predicate abstraction, LTL model checking, and so on. The implementation of the techniques described in this Thesis required to extend some nuXmv add-ons, as well as the software interface part using those new functionalities.

As described in Part II, the problem of minimal cutsets computation is an instance of parametric model checking. Therefore, the integration of this functionality required to extend the nuXmv add-on *param*, which is the one devoted to solve the parameter synthesis problem. In fact, the algorithms described in Part II come into play when the problem is configured to solve an instance with a set of parameters that satisfy the monotonicity assumption.

xSAP exposes the compute fault tree functionality as minimal cutsets computation, and it translates this problem to a parameter synthesis call for nuXmv. xSAP supports the definition of the system faults as input or state variables, thus they have different semantics. In case of a state

variable, its semantics expresses whether or not a fault has occurred in the past (from the initial states). Differently, defining a fault as an input variable requires xSAP to generate a fresh history state variable that implements the behavior "once fault" (similar to the history variables for the BDD case).

25.2 Contract-Based Safety Assessment

An OCRA system decomposition is defined with a hieararchy of components, linked between each other through input and output ports. Each component exposes a contract that expresses which behavior, on the output ports, the component is going to guarantee if a set of assumptions, on the input ports, are satisfied. Checking the validity of each contract refinement is thus a paramount analysis for an OCRA system decomposition. As described in Part III, the contract refinement checking is based on LTL satisfiability that can be reduced to an LTL model checking problem defined over a universal model i.e., a model that accepts all traces. Extending this concepts to the contract-based safety assessment, it results that each dependency analysis can be performed via minimal cutsets computation on a universal model. The software component that exposes this functionality to OCRA is part of xSAP, in fact it implements the core functionalities to solve the minimal cutsets computation problem.

25.2.1 Fault Tree Analysis on leaf implementation

An OCRA system decomposition is defined as a hierarchy of components, where the leaf ones can be refined with an SMV model implementation. The techniques described in Part III allow for combining the Fault Tree carried out on the hierarchical decomposition with the one resulting from the analysis of the implementation.

The integration between contract-based safety assessment and fault tree analysis on the implementations needs to be compliant with the philosophy of maintaining nominal and extended model disjoint. While for the analysis at contract level this requirement is implicit in the technique, for the FTA on the implementations, such integration requires reliance on automated model extension techniques. Due to this fact, the contract-based safety assessment functionality in OCRA accepts, in addition to the system decomposition, also a mapping file for the leaf components, and a fault extension information (FEI) file describing the fault injection pattern for each implementation. The algorithm combines all information, and generates a fault tree for each top level contract defined over basic implementation faults, or leaf components contracts in case the SMV implementation is not provided. The output formats of the contract-based safety assessment cover all the ones supported by the main Fault Tree manipulation softwares such as FT+, OpenFTA, XML, and CAFTA (Computer Aided Fault Tree Analysis System). Moreover, the hierarchical fault tree is represented as a formula in SMV format, which allows for automated analysis between different FTs like inclusion, and equivalence checking.

25.3 Redundant Architecture Analysis

The techniques proposed in Part IV are tailored to analyze the reliability of redundant architectures, defined as a set of components with input and output ports. Due to this fact the tool implementing the reliability analysis builds upon OCRA, exploiting its component-based architectural language.

This integration requires the OCRA system definition to be compliant with some modeling constraints that allow the reliability analysis techniques to be applied. More specifically, the tool-set is designed in order to take as input an OCRA system definition, and a map file that links

leaves components with redundant modules implementation. Given these inputs, the tool will then generate a new OCRA file, with its relative map, describing the miter construction. Once the miter has been constructed, the process relies on existing OCRA and xSAP analysis verification techniques, thus this approach does not require the user to get used to different software interfaces.

As shown in Part IV, the architecture analysis can be performed with two different techniques: concrete and modular abstraction. The implementation upon OCRA masks the details needed to apply the approach based on predicate abstraction, and it only exposes a parameter that allows the user to choose between the two techniques. The generation of stages, and their caching is managed by OCRA. The abstraction phase relies on nuXmv that in this case has been extended to support uninterpreted functions. Such level of automation has been achieved via the definition of an implementation library that contains both concrete and abstracted modules. The library currently contains 78 different modules, and the contraints for its extension are listed in the tool documentation.

26

Comprehensive Safety Assessment Process

26.1 Software Architecture

The techniques described in this Thesis extend the pre-existent tools, and refine the formal analysis process. Figure 26.1 shows how each Part of this Thesis contributed to extend each different formal analysis tool. Comparing this representation with the pre-existent one in Figure 24.1, now OCRA has a dependency with xSAP for the contract-based safety analysis technique, and the Reliability Analysis part relies on OCRA, xSAP, and nuXmv.

26.2 Integrated Process

In this Thesis we extend the pre-existent contract-based refinement with redundant architecture analysis and safety assessment. The direction of how those techniques should be integrated follows the design process defined in the ARP4754A [108].

The V model with safety assessment process prescribes the application of the safety analysis as an extension to the validation and verification

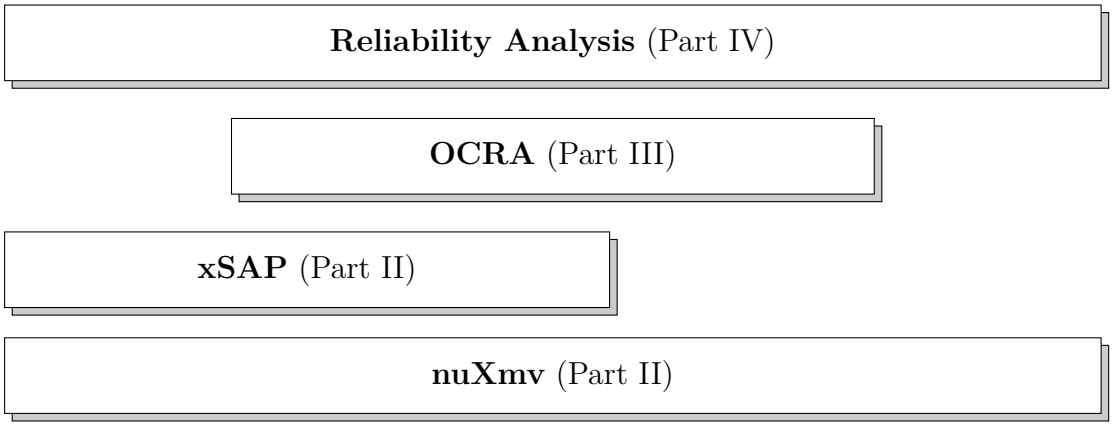


Figure 26.1: Software Architecture

phases. Thus, the integration of the techniques for redundant architecture analysis and contract-based safety analysis have to be compliant with this view.

The original process of the contract-based design implemented in OCRA [58, 54] covers the preliminary architecture and modules design phases of the development process. The former analyzes a system decomposition enriched with contracts by checking the consistency of the system refinement. The latter verifies that the leaf implementations (defined as state machines) obey the pre-defined contracts. This formal process covers part of the phases defined in ARP4754A. Figure 26.2 provides a schematic representation of the resulting process that combines pre-existent contract-based refinement with the techniques described in Part IV and III. The high level process is divided in the following phases:

1. *System Architecture Design and Optimization*: it concerns the definition of the system architecture from high level design to modules definition.

   (a) *Preliminary Architecture Design*: this phase defines the high level architecture, and extends it with system hazards definition on its

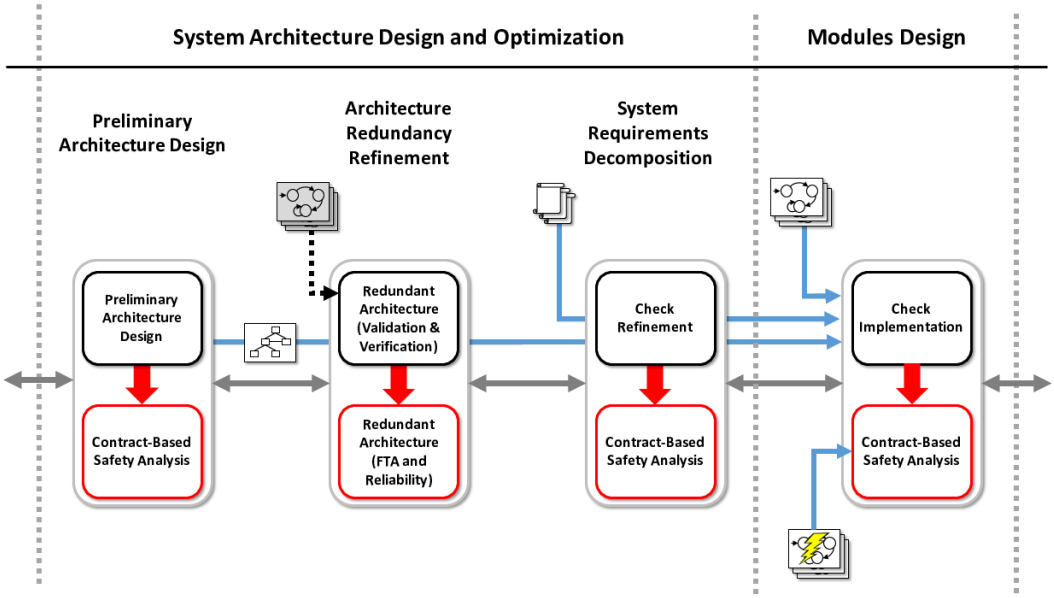


Figure 26.2: Comprehensive Formal Development Process

safety assessment counterpart.

(b) *Architecture Redundancy Refinement*: it elaborates on the redundancy aspects by refining the result of the previous phase.

(c) *System Requirements Decomposition*: in this step, the requirements obtained from the previous phases are decomposed at the modules level.

2. *Modules Design*: it concerns the architectural modules design.

The techniques introduced in Part IV aim at extending the process by providing the support during the definition of the system architecture. In particular, those techniques allow the system engineer to revise the architecture while preserving its interface and behavior. The provided analyses cover the validation and verification part (i.e., black border square in Figure 26.2), as well its safety analysis extension (i.e., red border square) with fault tree and reliability analysis. The artifacts produced in those phases are a summary of the property validation and verification (i.e.,

counterexamples), fault trees (as Minimal Cutsets), and reliability functions. Moreover, the redundant architecture analysis has been designed to extend seamlessly the pre-existent tool chain, by supporting an OCRA system decomposition as input. The library of components implementations (black dashed arrow in Figure 26.2) represents a dependency of the tool, and not a user defined input.

The contract-based safety analysis technique, described in Part III, is designed to be a complementary analysis to OCRA check refinement and check implementation. Thus, the contract-based safety analysis extends these steps of the design by computing a set of hierarchically organized fault trees out of a system decomposition enriched with contracts. At the modules level, the analysis requires, in addition to the previous steps, also a set of leaf components implementations with their fault injection directives. The contract-based safety analysis extends the design at both architectural and modules level with their counterparts on formal safety analysis.

27

Conclusion

In this Part we have described the pre-existing verification platforms and how they were have been extended in order to implement a coherent tools architecture. The resulting platform allowed for an extension of the previous verification process, while preserving the original methodology of relying on a single model of the system. In fact, the techniques integrate seamlessly with the pre-existing ones, while preserving the main philosophy of the model-based safety assessment of relying all analyses on a single formal model of the system.

The coherence in the extension of the formal verification process is a very important aspect when the new techniques are intended to be applied in practice. In fact, the usage of multiple, and possibly disjoint, formal models is highly error prone due to the fact that they are required to be kept properly aligned. Moreover, the user (e.g., the engineer) is not usually an expert in formal methods, and he should concentrate on a limited set of modeling formalisms. In Part VI we show how the methodology introduced here can be applied to several real-world case studies.

Part VI

Case Studies

Introduction

The aim of this Thesis is to develop a set of techniques able to aid the development of safety critical systems. The achievement of this target requires to go beyond a mere scalability evaluation, and analyze the proposed technology from multiple perspectives. In particular, our main concerns are in the direction to guarantee i) *conformability*, if the extension is compliant with the pre-existent process; ii) *expressivity*, whether our techniques are able to describe real systems; iii) *scalability*, if current (mid range) computational power is able to provide a reasonable level of performance. In order to validate the achievement of those targets, the techniques of minimal cut-sets computation (Part II), contract-based safety analysis (Part III), and redundant architectures analysis (Part IV) have been applied on a wide set of real-world case studies coming from the collaboration with leading edge partners such as the National Aeronautics and Space Administration (NASA) and the Boeing company. In particular, the case studies discussed in this Part are:

- *The Triple Modular Generator* (Chapter 28): provides a small but detailed overview of the integration of the (monolithical) model-based safety assessment into a standard formal V&V process;

- *Automated Air Traffic Control Design Space Exploration* (Chapter 29): describes the results of the NASA project that applied formal validation, verification, and safety assessment process to analyze the 1600

possible function allocation in the next generation of the air traffic control system;

- *Reliability Analysis on Fly-by-Wire Architectures* (Chapter 30): provides an evaluation of Fly-by-Wire system architectures of Airbus A330 and Boeing 777, by applying the techniques of redundant architecture analysis;
- *Formal Design and Safety Analysis of AIR6110 Wheel Brake System* (Chapter 31): applies the contract-based safety assessment to five possible interpretations of the Wheel Braking System architecture.

28

Triple Modular Generator

The system design process needs to cope with the increasing complexity and size of systems, motivating the replacement of labor intensive manual techniques with automated and semi-automated approaches. Recently, formal methods techniques, like model checking, have become competitive in the task of automated verification and validation.

In this Chapter, we show how to apply model checking techniques to a significant industrial case-study: a high integrity power distribution system such as required in aerospace, electrical power distribution, or the microgrid. These systems are composed of a redundant and reconfigurable plant and a controller that must guarantee a high level of reliability. However, the exponential number of possible configurations that have to be handled by the controllers of these systems make it challenging to manually verify and validate the correctness of the entire system. To address this problem, we apply formal methods techniques to the validation and verification of a triple redundant power distribution system. We model the plant, including faults, formalize and validate the requirements, and expose an ambiguity in the original natural language requirements. Moreover, by using the formalization of the requirements, we synthesize a correct by construction property-based controller, without the need of enumerating all possible

fault configuration (as currently done with manual approaches). The whole process is supported by the xSAP toolset, which also includes automated safety analysis such as Fault Tree generation.

28.1 Introduction

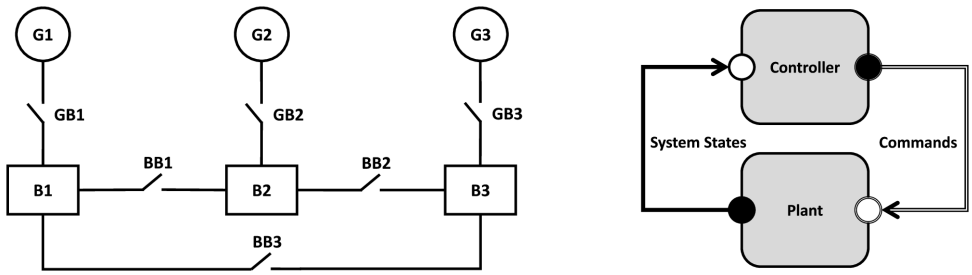
Providing continuous electrical power is a critical design requirement for systems in many industries. In aerospace, electrical generators on aircraft provide power to systems which must remain powered to ensure safe flight and landing. Redundant designs for the plant (generators, buses, circuit breakers) and controller must achieve reliability targets and ensure correct operation for all normal and non-normal configurations and failure combinations. The evolution to a More Electric Airplane (MEA) [78], as illustrated by the two most recent Boeing airplane models, the 777 in 1995 and the 787 in 2012, makes reliable electric power even more critical. This has led to more complex architectures and control schemes, resulting in significant increases in analysis complexity.

These analyses include 1) comparison of alternative redundancy schemes, 2) verification of system requirements, 3) safety analysis such as Fault Tree, Failure Modes and Effects Analysis (FMEA), common cause analysis, and zonal analysis, and 4) assessment of the cumulative effects of failures on systems, crew, and users. Similar, though less stringent, analyses apply to spacecraft and other transportation systems (rail, automotive, marine).

In the electrical power industry, distribution systems are the part of the grid between the high-voltage transmission lines and the consumer. Often distribution systems are connected to multiple power sources, and provide redundant paths and architectures (e.g., secured feeder, double supply, radial, interconnected primary feeders) to improve reliability of power deliv-

ery. The incorporation of distributed generation (DG) sources, distribution automation, and "self-healing" grid are among the recent trends [71] driving the evolution of distribution systems towards more complex system designs. To cite another example, microgrids [80] are emerging electrical power industry systems architectures that combine local, independent control of power generation, low voltage distribution, more fine-grained load management (load shedding), and the ability to operate connected or disconnected (islanding) from the main grid. All these emerging capabilities require monitoring and controlling DG sources/loads, diagnosis and isolation of defective portions of the grid, in order to provide very high levels of availability. Similar consideration to those of the aircraft industry apply to design, verification, operational control of these systems, and comparison of redundancy schemes. In contrast to aerospace, requirements for safety analysis are less stringent and fewer sensors mean that reconfiguration must be done with incomplete information, altering requirements for the controller.

In this Chapter we introduce an industrial case-study of a redundant on-board power distribution system with reconfiguration policy. The objective is to verify and validate the requirements expressing the expected behavior and provide a formal implementation of the controller that meets such requirements. The case-study was chosen to be comparable in complexity to the high voltage power systems of many previous generation aircraft, although details of the architecture may vary. For such systems the verification of power transfer can be performed manually by considering how to get to a legal configuration for all combinations of $0/1/2$ failures of the different components. This requires checking an exponential number of configurations against the reconfiguration requirements. Such an analysis is possible using manual techniques, but would be time consuming and error-prone. The interest of industry in applying formal methods



(a) Plant (b) Controller-Plant interaction

Figure 28.1: Redundant Power Distribution System representation

for such a task is driven by the evolution of the electrical system for MEA aircraft, which tend to have more generators and a more highly redundant distribution system, making these manual analyses highly labor intensive.

The case-study is composed of a plant representation and a set of informal requirements that express the expected behavior of the overalll system (plant, controller). The purpose of this work is to describe a comprehensive formal methods approach that can be applied to the formalization of the requirements (Section 28.3), the modeling of the system (Section 28.4), and the validation of the requirements and verification of the system (Section 28.5). The approach is supported by the xSAP toolset, which provides a broad set of techniques covering all the steps of the formal assessment.

28.2 Informal Problem

Figures 28.1 shows a triple redundant power generation and distribution system. The generators (G1, G2, and G3) provide variable frequency AC power to one or more buses (B1/B2/B3), depending on the state of the circuit breakers (GB1/GB2/GB3 and BB1/BB2/BB3). The controller sends commands to the generator (on/off), circuit breakers (open/close), and has sensors that report on the state of the components (Figure 28.1b).

The desired operation of the system is based on the physical properties of electrical power systems, and the level of reliability required by the application. In this example we assume that the required reliability can be provided by a Triple Redundant configuration (a.k.a. Triple Modular Generators) as shown in Figure 28.1a, with resiliency to all single and dual generator/circuit breaker failures. The requirements are derived from the physical properties of the system and the desired behavior of the overall system, and they are listed an described, respectively, in Table 28.1 and Table 28.2

The requirement *PR1* (see Table 28.1) describes that the physical system can not have fictitious current i.e. when generators are turned off all the buses must be unpowered. *PR2* and *PR3* define the behavior in case of short circuit: the former expresses the propagation of short circuit between different buses, while the second one describes the lack of fictitious short circuits. The buses break down if they receive power from more than one generator at the same time (*PR4*).

The controller requirements expressed in Table 28.2 impose that no bus may be connected to more than one power source at the same time (Requirement *CR1*), considering that the AC generators may be different. The controller behavior regarding the least number of powered buses are covered in *CR2* and *CR3*. A source to bus prioritization scheme determines the selection of power sources for each bus (Tables 28.3 and 28.4, *CR4*). The system has to be resilient to any single or dual failure, and this constraint is expressed by the requirement *CR5*. Moreover, only two generators should be on, unless the third is required to compensate for a failure (*CR6*). The requirement *CR7* expresses the controller reaction in case of bus short circuits.

ID	Description
PR1	If no power source is on, then all buses are unpowered.
PR2	Short circuits propagate between different buses if the buses are connected.
PR3	If no bus receives a short circuit, then no bus shall be in a short circuit state.
PR4	If a bus is connected to two different power sources, it breaks down.

Table 28.1: Physical System Behavior

ID	Description
CR1	No bus shall be connected to more than 1 power source at any time.
CR2	If any power source is on and there are no bus shorts, then all buses will be powered.
CR3	If any power source is on and there is a bus short, then at least one bus will be powered.
CR4	Bus power source priority and source-to-bus path priority schemes will be respected at all time (see Tables 28.3 and 28.4).
CR5	Any single/dual component failure shall not cause other system requirements to be violated.
CR6	Never more than two generators are on, unless required in case of failures.
CR7	Shorted buses shall be isolated from generators and other buses.

Table 28.2: Controller Requirements

28.2.1 System Faults

Each component represented in Figure 28.1a can fail to operate correctly, and in this study we cover a representative set of failures:

- Generators can fail off, permanently. When this fault occurs the internal state of the generator is stuck at *OFF* permanently;
- Circuit breakers can fail open and closed, transiently. The internal state of the circuit breaker is stuck at *OPEN* or *CLOSED* but the fault is not permanent (i.e. it is transient), so that the component

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

Table 28.3: Bus power source priority

Source to bus paths	Priority	B1	B2	B3
G1	High	—	BB1	BB3
	Low	—	BB3-BB2	BB1-BB2
G2	High	BB1	—	BB2
	Low	BB2-BB3	—	BB1-BB3
G3	High	BB3	BB2	—
	Low	BB2-BB1	BB3-BB1	—

Table 28.4: Source to bus path priority

can go back to the nominal behavior;

- Buses can reach a short circuit state, transiently. This condition can affect the other buses or generators if they remain connected together. The typical response to a short circuit is to isolate (disconnect) the part of the circuit experiencing the short.

28.3 Formalization of the Requirements

As described in Section 28.2, the requirements are split into two subsets: one that defines the plant validation laws, and one that concerns the verification of the controller. Thus, an important step when dealing with verification and validation is to provide a formal and unambiguous interpretation of the requirements.

```
1 INVARSPEC NAME
2 PR1 := ((P.G1.is_off & P.G2.is_off & P.G3.is_off) ->
3         (!P.B1.is_powered & !P.B2.is_powered & !P.B3.is_powered));
4
5 INVARSPEC NAME
6 PR2 := short_circuit_B1_to_B2 & short_circuit_B1_to_B3 &
7         short_circuit_B2_to_B1 & short_circuit_B2_to_B3 &
8         short_circuit_B3_to_B1 & short_circuit_B3_to_B2;
9
10 INVARSPEC NAME
11 PR3 :=  (P.count_ko_Bs = 0) -> (P.count_short_circuit = 0);
12
13 INVARSPEC NAME
14 PR4 :=
15  count(P.B1_poweredby_G1, P.B1_poweredby_G2, P.B1_poweredby_G3) > 1 -> P.B1.is_broken &
16  count(P.B2_poweredby_G1, P.B2_poweredby_G2, P.B2_poweredby_G3) > 1 -> P.B2.is_broken &
17  count(P.B3_poweredby_G1, P.B3_poweredby_G2, P.B3_poweredby_G3) > 1 -> P.B3.is_broken;
```

Figure 28.2: Validation of Plant model (SMV language)

This section provides the formal interpretation of the requirements described in natural language, with reference to the plant representation defined in Figure 28.1. The formal requirements are expressed via invariant properties that must hold for every state of the formal model.

28.3.1 Plant Validation

Table 28.1 defines a set of requirements for physical properties that the formal model must satisfy.

Requirement PR1 represents a basic law: when all generators are turned off no buses will be powered. This is a straightforward physical behavior, however the modeling of a system with circular connections can easily induce into unwanted behaviors. This requirement allows us to check the absence of fictitious current. i.e. The bus $B1$ in Figure 28.1a is powered if it receives power from any of its input, and propagates that energy to its other ports. However, if $B1$ receives power from $B2$, $B2$ receives power from $B3$, and $B3$ from $B1$, the system reaches the unrealistic condition where all buses are powered without a generator being on.

Requirement *PR2* expresses the fact that short-circuits propagate between two buses if they are connected with a circuit-breaker. The formal-

ization of this requirement is divided into six parts (i.e. they are 3 + 3 due to connection symmetry), one for each possible propagation pattern, as described in Figure 28.2. Each sub specification "$short\_circuit\_B_i\_to\_B_j$" is in the form "$B_i.is\_short\_circuit \wedge BB_j.is\_closed \rightarrow B_k.is\_short\_circuit$" and represents one direction.

Requirement *PR3* is similar to the physical requirement *PR1*, however it checks whether it is possible to obtain circular, and fictitious, short-circuits. In Figure 28.2 the invariant *PR3* expresses that if no bus is in short-circuit operational mode (i.e. "$count\_ko\_Bs$"), then no bus can be in a short-circuit state (i.e. "$count\_short\_circuit$"). The key aspect when modeling this property is that "$B_i.mode = ko$" (only) implies that "$B_i.state = short\_circuit$", but not the other direction. In fact, the former represents the condition when the bus is in a short-circuit failure, while the second expresses that the bus is working correctly, but it is linked to a component that is in a short-circuit state. This aspect allows us to distinguish between these two scenarios and express the requirement *PR4*.

The specification *PR4* expresses that if one bus is powered by more than one generator, it breaks down. This is formalized in by using the set of formulae "$B_i\_poweredby\_G_j$" that evaluate to True if at least one path is active from B_i to G_j (see Table 28.4). If more than one generator is connected to one bus, expressed by the formula "$count(...) > 1$", then that specific bus reaches the "broken" state i.e. "$count(...) > 1 \rightarrow B_i.is\_broken$".

28.3.2 Controller Verification

The invariant *CR1*, as in Figure 28.3, is based on the same set of formulae "$B_i\_is\_over\_powered$" which are defined in terms of "$B_i\_poweredby\_G_j$" as per in *PR3*, but in this case the *CR1* formula is True only if the controller is able to avoid to link more than one generator to one single bus.

The requirement *CR2* expresses that the controller must guarantee the

```
1 INVARSPEC NAME
2 CR1 := !P.B1.is_over_powered | !P.B2.is_over_powered | !P.B3.is_over_powered;
3
4 INVARSPEC NAME
5 CR2 := (CR5 & (P.short_circuits = 0)) ->
6        ((P.G1.is_on | P.G2.is_on | P.G3.is_on) -> powered_buses = 3);
7
8 INVARSPEC NAME
9 CR3 := CR5 -> ((P.G1.is_on | P.G2.is_on | P.G3.is_on) -> powered_buses > 0);
10
11 INVARSPEC NAME
12 CR4 := (CR5 & (P.powered_buses = P_comp.powered_buses)) ->
13        ((P_comp.score_path_B1 >= P.score_path_B1) &
14         (P_comp.score_path_B2 >= P.score_path_B2) &
15         (P_comp.score_path_B3 >= P.score_path_B3));
16
17 DEFINE
18 CR5 := (P.count_faults <= 2);
19
20 INVARSPEC NAME
21 CR6 := (P.count_faults = 0)->
22        (((P.powered_buses = P_comp.powered_buses) & (P.Gs_on = 3)) ->
23        (P_comp.Gs_on >= P.Gs_on));
24
25 INVARSPEC NAME
26 CR7 := (avoided_sc_B1_to_B2 & avoided_sc_B1_to_B3 & avoided_sc_B2_to_B1 &
27          avoided_sc_B2_to_B3 & avoided_sc_B3_to_B1 & avoided_sc_B3_to_B2 &
28          avoided_sc_B1_to_G1 & avoided_sc_B2_to_G2 & avoided_sc_B3_to_G3);
```

Figure 28.3: Formalization of Controller's requirements (SMV language)

functionality of all buses only if there are no short-circuits. This requirement must be integrated with the precodintion *CR5* (see the "DEFINE" in Figure 28.3), which constrains the behavior only in case there are no more than 2 faults. The invariant *CR2* in Figure 28.3 represents the formal interpretation of its informal counterpart.

The difference between *CR2* and *CR3* is the constraint on short-circuits: the first imposes their absence, while the second admits them. The formalization of *CR3* is quite simple, but in this case, if the requirement *CR5* is obeyed, the controller has to guarantee that at least one bus is working correctly.

CR4 defines an optimization requirement: it expresses that the source to bus path priorities have to be respected at all times, which means that the controller has to select the best possible configuration according to Tables 28.3 and 28.4. The formalization of *CR4*, as in Figure 28.3, is performed via an additional plant component called "*P_comp*". This ad-

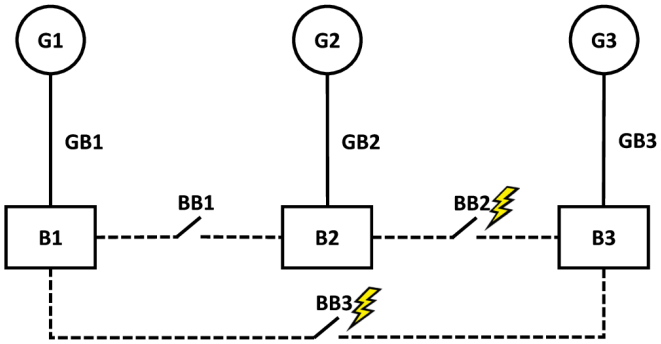


Figure 28.4: Plant configuration with double faults

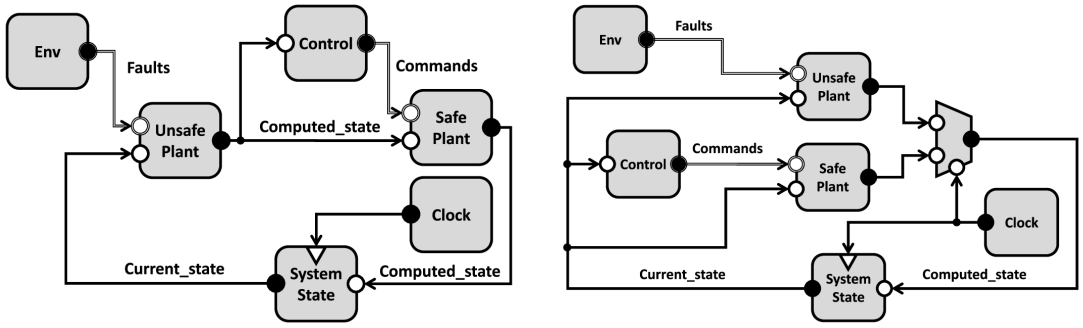
ditional plant is used as a comparison for the controller operations, and it is free to choose every possible commands considering that it receives as input the same states and operational modes of "P". This approach allows us to check, via an invariant property, if there exists a command configuration that is better than the one provided by the controller. The formal representation of *CR4* represents the optimality via the formulae "$score\_path\_B_i$" that evaluate to an integer number such that the bigger the value is, the lower is the priority. The premise of the *CR4* implication considers the case where the two plants have the same number of powered buses, in addition to the integration with *CR5*.

The approach used in the formalization of *CR6* relies again on the comparison parallel plant "P_comp" since also this requirement expresses an optimization constraint. In fact, the sentence "never more than two generators on" is checked by stating that if the controller chooses to turn on 3 generators, as expressed by "$P.Gs\_on = 3$", then it is not possible to provide a different reaction that uses fewer generators; the invariant "$P\_comp.Gs\_on \geq P.Gs\_on$" expresses this constraint.

Requirement *CR6* is essentially composed of two parts: the first one, represented by "never more than two generators on", which is unambiguous, while the second one represents a classic example of ambiguity in natural

language, since the expression "unless required in case of failures" can have more than one interpretation. The formal translation of *CR6*, represented in Figure 28.3, considers the second sentence as "$P.count\_faults = 0$", which essentially means that there are no faults. However, this part could also be interpreted as the invariant "$P.count\_faults \leq 2$", meaning that there are less than three faults. The interesting implication of this last interpretation is that *CR6* requirement, combined with *CR4*, leads to a contradiction. Figure 28.4 represents this case considering a configuration when BB2 and BB3 are stuck at open, and according to requirement *CR4* the controller selects the highest priority paths for both B1 and B2, while for B3 the only possible choice is via GB3. Figure 28.4 shows that the controller reaction is to turn on all generators, but the closing of BB1 and opening of GB2 would allow using only two power sources. In other words, this example shows that it is not possible to obey both *CR4* and *CR6* if, in *CR6* the sentence "unless required in case of failures" is interpreted as "$P.count\_faults \leq 2$".

CR7 expresses the ability of the controller to avoid short-circuit propagation. In fact, if two buses are connected and one of them is in short-circuit, then both will be operating in the same operational mode. Thus, the controller has to avoid this behavior, by opening the suitable circuit breakers whenever possible e.g. when they are not in stuck at closed. The constraint *CR7* is based on the set of formulae named with "$avoided\_sc\_B_i\_to\_B_j$" that check if the controller avoids a short-circuit propagation from B_i to B_j, under the condition where the circuit breaker between B_i and B_j is not operating in a fault mode.



(a) Instantaneous propagation (b) Interleaved propagation

Figure 28.5: Plant configurations

28.4 Formal Model

28.4.1 Plant Modeling and Model Extension

The plant and the controller are modeled independently and their combination is then used for providing validation. In particular, as shown in Figure 28.5, we provide two possible configurations for their combination: Instantaneous and Interleaved.

In the *instantaneous* configuration, the controller ("Control" in Figure 28.5) can react much faster than the plant. Upon the occurrence of a fault, the controller is able to deliberate on the commands to send to the plant much faster than the effects of the fault can manifest. Therefore, the plant never reaches an unsafe configuration. In a sense, we can imagine that the controller is able to foresee the faults and prevent the system from going to an unsafe configuration.

In the *interleaved* configuration, the controller can only react to the plant being in an unsafe state. In this situation, we assume that the plant is able to go to an unsafe state for a short time (one unit of time) and that this will not violate any requirement due, for example, to some tolerance of the components. The controller will then bring the system back into a

safe state.

More in detail, the model of the system is divided in several components (see Figure 28.5). All the components are memory-less, and the state of the system is stored in the *System State* memory. The most interesting components are the controller, and the two copies of the plant: *Safe Plant* and *Unsafe Plant*. These two copies of the plant are identical except for the inputs they receive. The unsafe plant takes as input the state of the system, and a (possibly empty) set of faults from the environment ("Env" in Figure 28.5), and provides as output a configuration of the system that is (potentially) unsafe. Similarly, the safe plant takes as input the state of the system and the commands from the controller, and provides as output a (potentially) safe configuration of the plant.

In the instantaneous case, the output of the unsafe plant is fed directly to the controller, that is able to react instantly and provide its outputs to the safe plant. The position of the memory after the output of the safe plant, ensure that (if a good controller is available) at each time step, the system state (as stored in the memory element) is safe.

In the interleaved case, the outputs of both the safe and unsafe plants are fed to a multiplexer. This multiplexer, which is driven by the system clock, alternates the output between the two plants at every time step. In this way, the controller takes the outputs of the unsafe plant and computes the necessary commands for the safe plant for the next time step. Note that in the interleaved configuration the traces are longer than in the instantaneous configuration since commands and faults are interleaved (see Figure 28.6). However, this style of modeling can be considered closer to reality, since the controller reacts to observations on the system, instead of foreseeing the behavior of the plant.

Note how the important building-blocks (Controller, Safe Plant, Unsafe Plant) are reused in both configurations. This provides a way of studying

(a) Instantaneous Trace (b) Interleaved Trace

Figure 28.6: Comparison of traces

both configurations while avoiding doubling the modeling effort.

The properties that validate the controller take into account this aspect, and the formal model has to explicitly express when a state is "safe" or "not safe", by distinguishing between time frames when the controller does and does not have the capability to react. This condition defines that there may be some time frames when the system is operating outside the requirements while waiting for a controller reaction. In order to prevent wrong results in the verification, we need to explicitly express that each property holds only in a "safe state".

28.4.2 Controller Development

The objective of the controller is to command the plant so that it avoids unsafe conditions, and minimize the impact of component failures. In order to reach this objective, the controller reads an input configuration (e.g., the current one), and provides a set of signals that can change the state of the plant into one that is considered "safe". The difference between a safe, or acceptable, state and an unacceptable one is defined by the requirements. The requirements in this case-study express a set of conditions that are independent of the history of the signals that are received by the controller, and due to this fact, such specifications allow for the definition of a "memory-less" component. Therefore, we can define a controller by analyzing each possible configuration that is intended to be managed, and explicitly define the reaction to each undesirable condition. However, this approach is error prone and time consuming, considering that the set of all

possible configurations is in the order of $2.3 * 10^4$ (i.e. the controller has to manage: 2^3 configurations for the generators, 3^6 for circuit breakers, and 2^3 for buses, and the overall result must be divided by 2 in order to remove symmetric configurations). The requirements defined in Section 28.2 provide a bound on the fault cardinality that must be considered by the controller. This bound, as expressed by requirement *CR5*, is set to 2; based on this, it is possible to define a controller with an explicit representation of 165 configurations. In fact, if we call F_i the number of possible fault modes for the component i, then number of all the possible configurations with 2 faults is $\sum_{i=1}^{n} \bar{F}_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \bar{F}_i * \bar{F}_j$.

Another possibility is to define a property-based controller, which avoids explicitly defining the behavior for each possible configuration, and it is based on a set of formal properties that define the behavior needed to obey the requirements. Specifically, the challenge of this technique is to define a set of symbolic formulae such that their conjunction represents the acceptable states.

The property-based controller specifically defined below is composed of the following sections: preservation of bus functionality, avoidance of short-circuit propagation, prioritization of generator to bus paths, and minimization of power usage.

Preserving Bus functionality Each bus can be linked to any generator, and if one bus is connected to more than one generator, it becomes not functional. The controller needs to avoid this condition, and the formula that expresses this constraint is in the form "$count(B_i\_poweredby\_G_1$, $B_i\_poweredby\_G_2$,
$B_i\_poweredby\_G3) \leq 1$", where $B_i\_poweredby\_G_j$ becomes true when the commands are sent to the plant, according with its operational modes, link B_i to G_j by any of the possible paths.

Avoidance of short-circuit propagation Similarly to the preserving of bus functionality, this section is necessary to avoid the propagation of short-circuits. The constraints are in the form "$B_i\_is\_short\_circuit \rightarrow cmd\_B_i\_unlink$", which expresses the priority to unlink the bus B_i from its neighbors, both the closest generator and adjacent buses, if it is in a short-circuit operational mode.

Prioritization of generator-to-bus paths The requirements define an explicit ordered set of the possible paths between buses and generators. This constraint is expressed via three case condition, one for each bus. Each condition is in the form "$can\_do\_B_i\_poweredby\_G_J\_\{L, U, R\} \rightarrow cmd\_B_i\_poweredby\_G_j\_[L, U, R]$", where "$can\_do\_B_i\_poweredby\_G_J\_\{L, U, R\}$" expresses the possibility to enable the path between B_i and G_J through either the left port L of B_i, the upper port U, or via R that is the right one. Similarly, $cmd\_B_i\_poweredby\_G_j\_\{L, U, R\}$ defines the commands needed to perform the action that is expressed in the premise of the implication.

Minimization of power usage The minimization of power usage is one of the targets that the controller has to reach. For this case-study no more than two generators may be turned on, unless required in the case of failure(s). The controller implements this constraint via the prioritization of generator to bus paths constraints, in addition to a simple formula defining that, when the circuit breaker GB_i is open then a cmd_off must be sent to the generator G_i.

28.5 Verification and Validation

The formal properties defined in Figure 28.2 and 28.3, defined as invariants, can be checked using the nuXmv via pure BDD reachability analysis. The time needed to perform this task is in the order of 2 seconds. The results provided by the model checker were positive for all properties, both for plant and controller requirements.

The requirements *CR2* and *CR3* defined in Section 28.3 express the level of bus functionality that has to be guaranteed in case of single and double faults. The verification of these requirements can be performed via invariant checking, but Fault Tree Analysis (relying on xSAP) is able to provide a more detailed description of the system behavior. Specifically, the fault tree that represents the conditions where the buses are not powered give us the motivation of why requirements *CR2* and *CR3* are satisfied.

Fault Tree Analysis for this model considers four different Top Level Events: with 1 or 3 buses that are not powered (i.e., the TLE name that ends with "_1" or "_3" in Table 28.5), and with or without short-circuits (i.e., labelled with "*_sc*" or "*_nsc*"). The Minimal CutSets computed with these Top Level Events meet the expectations. Table 28.5 summarizes the results of the Fault Tree Analysis by providing the number of Minimal CutSets ordered with respect to their cardinality, and such results can be explained as follows:

- TLE_sc_1: this TLE expresses the condition when at least one bus is powered, under the condition that short-circuit operational mode can affect the buses. Table 28.5 shows that the possible configurations are 3 of cardinality 1, that essentially represent each single short-circuit. The configurations of cardinality 3 and 4 express the conditions where multiple faults on generators and circuit breakers may force one or two buses to be unpowered;

- TLE_nsc_1: this result is similar to TLE_sc_1, but in this case there are no configurations of cardinality 1. It is important to remark that this result provides evidence that requirement *CR2*, combined with *CR5*, is obeyed due to the fact that the sets of configurations with cardinality 1 and 2 are empty;

- TLE_sc_3: in this case, the Minimal Cutsets express which are the configurations that lead to having all buses that are not working. Table 28.5 reports that the possible Minimal CutSets are 54 of cardinality 3, corresponding to the configurations where: i) 1 bus is in short-circuit and the neighboring switches are stuck at closed, ii) 1 bus is in short-circuit and 1 switch is stuck at closed, plus a generator that is stuck at off. iii) 2 buses are in short-circuit and a generator is stuck at off. Thus, the lack of configurations with cardinality 1 and 2 provides the evidence that requirement *CR3*, combined with *CR5*, is obeyed;

- TLE_nsc_3: if the TLE expresses the possibility of having all buses that are upowered, but without the ability to reach a short-circuit operational mode, the possible configurations are 8 of cardinality 3. Specifically, such configurations represent the condition where all the generators, combined with the circuit breakers GB_i, are not working.

28.6 Conclusions and Future work

In this Chapter we introduced an industrial case-study that represents a redundant on-board power supply with reconfiguration policy. The challenge was to verify and validate the requirements expressing the expected behavior and provide an actual formal implementation of the controller to meet such requirements.

Cardinality	Number of Minimal CutSets			
	TLE_sc_1	**TLE_nsc_1**	**TLE_sc_3**	**TLE_nsc_3**
0	0	0	0	0
1	3	0	0	0
2	0	0	0	0
3	14	14	54	8
4	12	12	0	0
≥ 5	0	0	0	0

Table 28.5: Fault Tree Analysis Results

The high voltage power systems of many previous generation aircraft are comparable in complexity to the plant shown in Figure 28.1. For such systems the verification of power transfer, for the conditions described in this study, can be done manually by considering how to get to a legal configuration for all combinations of 0/1/2 failures of generator/circuit breaker/bus, leading to up to 2^{12} configurations to be verified. Although possible, performing such an analysis manually would be time consuming and error-prone.

Evolution to a MEA has several key impacts on the electrical power system: generators must produce significantly more power, the system must have a higher reliability, and the system must be certified to a higher level of criticality. As a result the electrical system for MEA aircraft will tend to have more generators and a more highly redundant distribution system. The resulting configurations will typically have 2-3 times as many components, leading to a significantly more complex analysis. Assuming twice the number of components, the number of configurations that need to be analyzed is about $2^{24}/2$ (divided by 2 to account for symmetry), or 2^{12} times as many configurations as described in this case study. Clearly with a problem of this scale manual methods are no longer practical.

In addition to the high voltage portion of the power distribution system we have been discussing, aircraft also have low voltage distribution systems, typically with 28VDC and 120VAC buses, external (ground) power sources, a battery, and transformer/rectifiers that connect the high/low voltage segments of the overall power system [89]. The low voltage portion typically has at least as many components as the high voltage portion.

The analysis approach described in this study are targeted at the verification problem of the complete interconnected power distribution system. In addition to the benefits of verification, potentially there is a significant advantage simplifying how a complex controller is implemented in early phases of the design life-cycle. The verification and validation process described in this Chapter is supported by nuXmv and xSAP tools, which allowed us to perform model checking, as well as model-based safety assessment analyses, using a single software workbench.

As a future direction of our work, we need to assess if the formal verification and property-based controller development approaches hold up at the scale of MEA size interconnected power system, or if supplementary techniques are necessary (partitioning/abstraction/etc). We would also like the model and analysis tools to support other types of analysis that are performed early in the life-cycle, e.g., architecture trade studies and probabilistic analysis when varying the architectural displacement of each single component.

29

Automated Air Traffic Control Design Space Exploration

In response to the need to increase the capacity of commercial air space, NASA has been tasked with identifying and studying more automated solutions for air traffic control. When multiple ideas and designs are available, there is a need to map the design space in order to understand the big picture, and be able to understand the impact of design choices on the overall functionality and safety of the system. In this context, we are faced with a big design space, where a large number of different solutions are possible.

Reasoning on such complex and safety critical systems can clearly benefit from the use of formal methods techniques [124, 76, 119]. The goal of this work is to understand the trade-offs of placing separation assurance functions on-ground versus on-board or in a mixed-mode that switches between the two, in order to increase the capacity of the air space in a provably safe way. In collaboration with NASA Ames and NASA Langley experts, we define more than 1,600 instructive configurations. We analyze all of these configurations with formal tools and compare their merits. This results in significant insights on the features of the various configurations. We discuss how formal methods have been applied to support the exploration of this practically relevant design space, as part of an internal

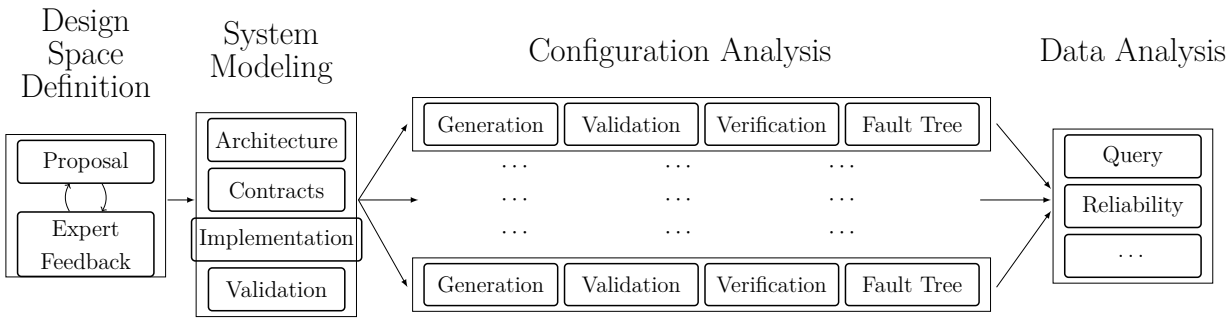


Figure 29.1: Process Overview

project of NASA Ames.

The activity can be summarized in four main phases, depicted in Figure 29.1: *Design Space Definition*, *System Modeling*, *Configuration Analysis*, and *Data Analysis*.

Design Space Definition. The stage was set by identifying precisely (yet informally [84]) the situations of interest, and by defining the modeling dimensions to capture them. Interacting with the domain experts we defined a structured design space that contained more than 1,600 configurations worth analyzing.

System Modeling. Performing the modeling of each solution independently would be too time-consuming (if not outright unfeasible). Plus each model needs to be properly validated, to make sure that it enjoys the expected properties. Furthermore, independent modeling would require a lot of maintenance effort to ensure all models remain aligned with each other and with NASA's most current designs, and to propagate changes and modifications uniformly among them. In order to manage these sources of complexity, we combined several ingredients. First, we used the architectural language of OCRA to separate the system architecture from the implementation of the single components. This allowed us to model each component in isolation, partitioning the effort, and minimizing the time required to validate changes in any component. Additionally, we can change

the implementation of a single component without impacting the rest of the system. Second, we used contracts to characterize each component. This allowed us to properly specify the interaction between components, and decompose the properties required for validation from each component into properties for its sub-components. Third, we used parameters to factor out multiple configurations into a single (although more complex) model. If two configurations require only marginal changes to an implementation, we capture these changes using parameters within the models. These techniques allowed us to automatically generate formal models for all of the configurations in the design space, with great confidence in their correctness and alignment. This phase was performed only once, without the need of repeating it for each configuration.

Configuration Analysis. Each model was verified against the properties of interest; in addition, techniques for safety assessment were applied to identify which configurations of faults lead to the loss of fundamental properties. The corresponding fault trees were automatically computed, thus providing additional information on the reliability of each configuration. Each configuration was instantiated and analyzed independently, exploiting the parallelism typical of modern HPC infrastructures, and significantly speeding up the analysis.

Data Analysis. The results were combined into a symbolically represented dataset, linking each configuration to its satisfied properties and fault trees. This dataset can be studied offline to automatically extract sets of configurations enjoying specific properties (e.g., absence of single points of failure). This dataset is particularly useful in such an explorative phase, since it describes the whole design space and can be studied to better understand it. Finally, we discussed a selection of interesting configurations with the domain experts at NASA, and identified well-known results [79, 74] as well as a novel one. In particular, we highlighted the

need for additional assumptions when dealing with changes in delegation of separation assurance from an aircraft to the ground (e.g., in case of a request for backup).

The rest of the Chapter is structured according to the process described above (Figure 29.1). Section 29.1 provides some necessary background. Section 29.2 describes the modeling approach used in this work. Sections 29.3 to 29.7 describe each phase of the process in greater detail. Notable results extracted from the analysis are discussed in Section 29.8. Related works are discussed in Section 29.9, and Section 29.10 concludes with possible directions for future work.

29.1 Background Notions on Automated Air Traffic Control System

NASA is tasked with designing the next, more automated, air traffic control system for the United States. A major safety goal is to minimize Loss of Separation (LoS), resolve any such situations immediately, and never call upon collision avoidance. LoS occurs when two or more aircraft become too close to each other, i.e., they are below a defined safe distance of 1000 feet vertical and 5 nautical mile horizontal separation. If LoS is not resolved immediately, collision avoidance is necessary. The functional allocation question asks how which separation assurance (SA) capabilities to require and how to distribute the functions of the design in combination with a subset of these capabilities on top of a set of agents, in order to minimize the number of LoS and the use of collision avoidance techniques [84]. We consider the following agents, functions, and capabilities:

Functions:

- **Strategic Separation** addresses short-term conflicts from 20 minutes in the future down to 3 minutes out from a predicted LoS. Strategic

separation is implemented in software and can be running on a central computer on the ground, on-board individual aircraft, or some combination thereof. It uses the trajectories of each known aircraft in the airspace, detecting any conflicts, and outputting resolution maneuvers for any aircraft involved in conflicts.

- **Tactical Separation** addresses near-term conflicts predicted to occur less than 3 minutes in the future. It is also implemented in software running on either a ground computer, an on-board computer, or a combination thereof. Tactical separation must employ a different algorithm from strategic separation because the conflicts it addresses are more imminent and different details must be considered when generating resolution maneuvers.

- **Collision Avoidance** addresses possible collisions less than 30 seconds in the future. Its presence is required by Federal Aviation Administration (FAA) mandate, therefore, TCAS (and in the future ACAS-X), software runs on-board every aircraft, detects possible collisions using a transponder installed in the aircraft, and must operate totally independently from on-ground systems. A system safety objective is to never trigger collision avoidance.

Agents:

- **Self-Separating Aircraft (SSEP)** carry a separation assurance software on-board.

- **Ground-Separated Aircraft (GSEP)** rely on SA software running on a central on-ground computer transmitting to the aircraft.

- **Air Traffic Control (ATC)** Provides on-ground separation of GSEPs and, when needed, of SSEPs.

Capabilities:

- **ADS-B Out** (Automatic Dependent Surveillance-Broadcast Out) is required on-board all aircraft by FAA mandate by 2020; it broadcasts position information to ADS-B ground stations and other aircraft within transmission range.
- **ADS-B In** is optional by FAA regulations; it receives ADS-B broadcasts from ground stations and other aircraft.

Depending on who is in charge of what, and the available resources, we can describe different designs. Different designs will have different characteristics. Our goal is to provide some qualitative measure of the goodness of each solution along different dimensions. For example, in a scenario in which both GSEP and SSEP aircraft are involved, we might want to know whether a solution in which SSEPs perform both tactical and strategic separation on-board is "better" than a solution in which tactical separation is handled on-ground.

29.2 Formal Modeling for Comparative Analysis

An important aspect is then to define the right level of abstraction in order to guarantee that all the relevant aspects are taken into account. In the following sections, we detail what variables define the state of the system, how time passes, and how this influences the change in the state. Note that our analysis focuses on the protocol level and thus, in absence of faults, we assume each component implementation to be correct.

29.2.1 Trajectory Intentions and Conflict Areas

The basic information that is relevant for our analysis is the trajectories that aircraft intend to follow, and more specifically if their intention is

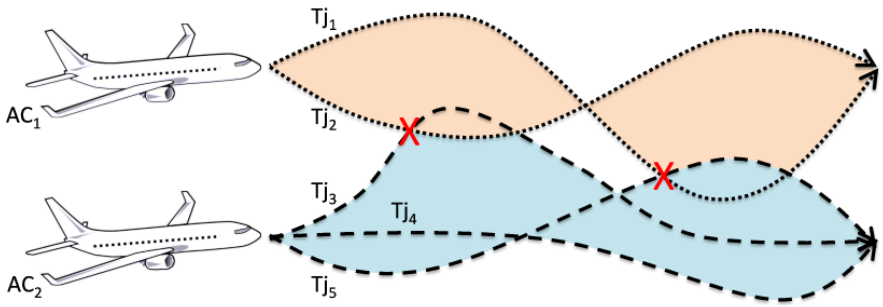


Figure 29.2: Conflict Areas abstraction

in conflict with other aircraft. The actual detail of the trajectories (i.e., the 3D position as a function of time) is not part of our model. In fact, we reason about the system at the architectural level, focusing on the interaction between the components rather than on their precise behavior. We are not interested in which specific trajectory an aircraft should follow to avoid a collision, but only in whether their intentions are in conflict or not. Therefore, we abstract away the detailed trajectory information by introducing *Conflict Areas* (CA). Intuitively, two aircraft are in the same CA, if their trajectories intersect in a given interval of time. In this way, we can abstract the problem of separation into the simpler problem of checking that two aircraft are not in the same conflict area. Figure 29.2 shows an example when two aircraft have to reach two separate destinations. In this example we consider Tj_1 and Tj_2 for AC_1, and Tj_3, Tj_4, Tj_5 for AC_2. Figure 29.2 shows that AC_1 and AC_2 are in the same CA if their intended trajectories are respectively Tj_1 and Tj_5, or Tj_2 and Tj_3. In all other cases, they are into different CAs, representing the absence of conflicts. CAs are used throughout our models anytime we talk about aircraft intentions and resolutions sent by controllers.

29.2.2 Time windows

Most scenarios in [84] divide the responsibility of the separation-assurance agents based on *time windows*. In particular, we consider four time windows: *Current*, *Near*, *Mid* and *Far*. They represent symbolically consecutive time intervals. Therefore, the trajectory intention of the aircraft define which aircraft are in the same CA in each time window, as defined in the previous section.

The Current window represents the immediate intention of the aircraft, i.e. within 30 seconds. This window is managed by *Conflict Avoidance* algorithms, e.g., TCAS, and is therefore the key to the definition of LoS: two aircraft are currently in LoS if they share the same conflict area in the Current window. The tasks of tactical and strategic separation are then mapped into the Near- and Mid-window (Tactical) and the Far-window (Strategic). If two aircraft share the same conflict area in the same window, we say that we have a *predicted* LoS.

In our model, the intention of aircraft is represented by assigning each airplane with a CA for each window. Figure 29.3 shows an example with two aircraft. In this example, the aircraft are in different CAs apart from the Far window. So, we have a predicted LoS in that time window.

Intuitively, the windows shift with the passage of time: the old Near information will became the new Current information (Figure 29.3), while the intention for the other time windows change according to the interaction among the agents. Therefore, if we manage to resolve all predicted LoS, e.g., in the Mid window, we will not have LoS. In order for conflicts to be detected and resolved, we need to take into account the communication between aircraft and the ATC and when it occurs. In the model, passing of time is divided into two main phases that alternate constantly: *communication* and *maneuvering*.

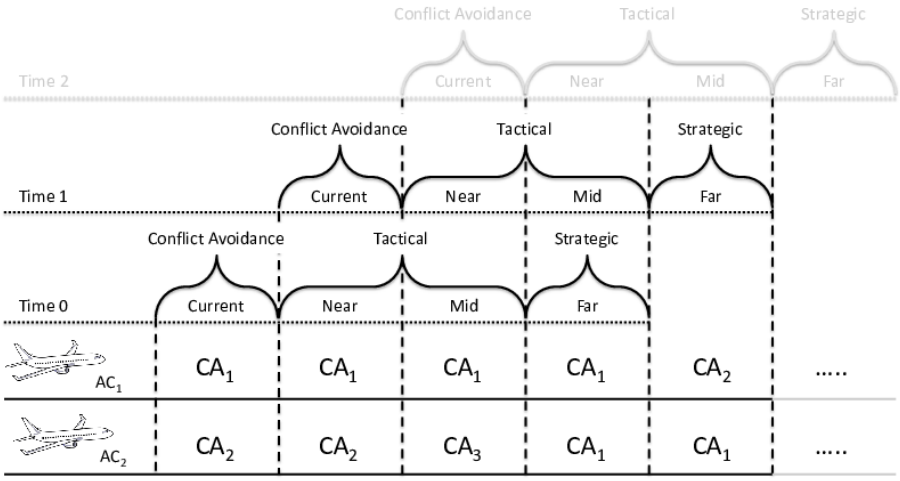


Figure 29.3: Near, Mid, Far windows, and their shifting

During the maneuvering phase, windows are shifted (Figure 29.3). During the communication phase, the different agents are able to exchange intentions and resolutions. For example, the aircraft is able to provide its intention to the ATC, and receive a suggestion for a new trajectory. We introduce a bound on the number of communications during this phase, in order to better understand whether multiple iterations between agents can improve the reliability of the system. This interleaving model may seem unintuitive. However, this choice is justified by reality since we can only apply a maneuver after deciding it, and it simplifies the modeling.

29.3 Design Space Definition

Our work started by considering several proposals from NASA's AFT Branch for different solutions for the Function Allocation for Separation Assurance [84]. These ideas were the result of considering several features and characteristics in a preliminary phase. Our first step was to identify and formalize the dimensions shared by different proposals, in order to define the design space. We derived six modeling dimensions that enable us

to capture different trade-offs:

1. SSEP Separation Agent
2. Aircraft Mix
3. Information Sharing
4. Burdening Rules
5. Communication Steps
6. ACDR Implementations

SSEP Separation Agent. A key difference between the solutions is who is in charge of performing separation for the SSEPs. We split this task into separation for the Tactical (Near and Mid) and Strategic (Far) windows. For each of these windows we define who is in charge of separating the SSEPs: the ground (ATC), the aircraft ($SELF$), or the aircraft with possible delegation to ground ($SATC$). If ground is in charge of separating the SSEP, the resolutions are computed by ground, and sent to the aircraft to apply them. If the aircraft is in charge of its own separation, computation of a resolution strategy happens on-board, possibly involving coordination between aircraft. The third case ($SATC$) captures the possibility for an SSEP to delegate its own separation to the ground. This is used to capture different situations such as backup in case of a fault, privileged traffic corridors, and transfer of responsibility in designated airspace regions. In the future, we expect other cases to be studied. For example, resolutions might be computed on-board but require approval from ground.

Aircraft Mix. We consider situations in which all aircraft are of the same type, and situations in which we have a mix of aircraft types. The same design can be analyzed without SSEPs, with the same number of GSEPs

and SSEPs, without GSEPs, or any option in-between. Each combination is indicated by the number of GSEPs and SSEPs, i.e., $\langle \#GSEP, \#SSEP \rangle$.

Information Sharing. We want to minimize required communication, because it adds time and complexity. Therefore, we need to understand what is the minimum amount of intent that aircraft need to share. We make two main distinctions: information sharing from *GSEPs to SSEPs* and from *SSEPs to ATC*. For each of these two information sharing pipelines, we consider scenarios from sharing no information (*None*) to sharing information concerning just the *Current* window, up to *Near* window, up to *Mid* window, or all the windows (*Far*).

Burdening Rules We might want to specify who has priority in a conflict between a GSEP and an SSEP. For example, we might say that SSEPs always have the right of way, and therefore the burden of maneuvering is on the GSEPs. Burdening rules define who should move when such a conflict occurs: 1) *Undefined*, 2) *GSEP*, 3) *SSEP*. If the burden is on the GSEP, then the conflict should be resolved by changing the trajectory of the GSEP. If the burdening rules are undefined, then each agent will arbitrarily choose a burdened strategy, and consistently apply it to every conflict.

Communication Steps. Due to delays in communication and availability of the networks, multiple rounds of communication might be needed during the *Current* time window in order to achieve an agreement among the agents. Instead of having only one communication round, we model for the maximum number of rounds that could realistically transpire, given minimum hardware latencies.

Name	Possible		Considered	
	Values	Size	Values	Size
SSEP TS SA	ATC, SELF, SATC	3	ATC, SELF, SATC	3
SSEP SS SA	ATC, SELF, SATC	3	ATC, SELF, SATC	3
Aircraft Mix	$\langle 4,0\rangle$, $\langle 3,1\rangle$, $\langle 2,2\rangle$, $\langle 1,3\rangle$, $\langle 0,4\rangle$	5	$\langle 4,0\rangle$, $\langle 3,1\rangle$, $\langle 2,2\rangle$, $\langle 1,3\rangle$, $\langle 0,4\rangle$	5
GSEPs to SSEPs Info	None, Current, Near, Mid, Far	5	Current, Far	**2**
SSEPs to ATC Info	None, Current, Near, Mid, Far	5	Far	**1**
Burdening Rules	Undef, GSEP, SSEP	3	Undef, GSEP, SSEP	3
Com Steps	1, 2,	2	1, 2,	2
ACDR Implementations	Simple, Asymmetric, Non-Receptive	3	Simple, Asymmetric, Non-Receptive	3
TOTAL		20,250		1,620

Table 29.1: Summary of possible and considered design dimensions

ACDR Implementations Each component of the system can be implemented in multiple ways. In this project, we considered different implementations only for the Airborne Conflict Detection and Resolution (ACDR) component. The simplest implementation of the ACDR computes a resolution without considering the behavior of the other aircraft ("ACDR Simple"). A more complex implementation, instead, will take into account how the other SSEPs are going to resolve the conflict, and use this knowledge to compute a resolution that is guaranteed to solve the current conflict ("ACDR Asymmetric"). Finally, the last implementation (called "ACDR Non-Receptive") is the one in which we declaratively enforce the assumption that conflicts will be resolved without specifying how, thus constraining the environment with a non-receptive specification [1]; this last option is useful to study the system behavior assuming a perfect ACDR.

Table 29.1 shows the possible dimensions defined during the first analysis, and yields a design space with more than 20,000 configurations. After some discussion, NASA domain experts proposed to exclude some configurations that were not interesting from the domain point of view. In particular, they suggested to fix the information sharing of the SSEPs, in order to provide all information (i.e., *Far*) and consider only the two extreme cases for the information shared by the GSEPs: *Current* and *Far*.

The reduced design space (right part of Table 29.1) obtained after applying these simplifications, yields a set of 1,620 configurations. These are the configurations taken into account in this work.

29.4 System Modeling

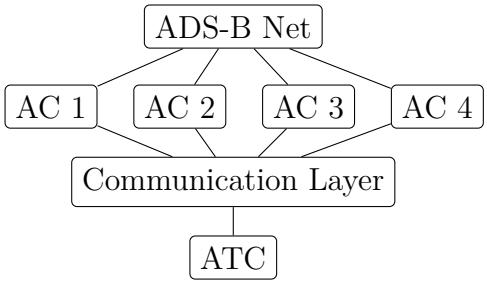


Figure 29.4: Model Architecture

The dimensions described in Table 29.1 are captured by defining a unified structure including all possible configurations. This structure is equipped with parameters and multiple implementations of the components, making it possible to model the whole system once, and then automatically generate any of the 1,620 possible instances. This reduces the modeling effort that is, in terms of resources, the most expensive part of the process. However, we need to pay particular attention to the validation of the model, to make sure that all expected behaviors are captured once parameters have been instantiated.

The general structure of the model is shown in Figure 29.4, and includes four aircraft, the ATC, and two different types of networks: ADS-B and Communication Layer. ADS-B is used only among the aircraft, while the Communication Layer is used between the aircraft and the ATC. This choice makes it simple to provide different characteristics to the two networks: faults, symmetry, amount of information, delays, etc. We always consider up to four aircraft instances. This is sufficient to capture

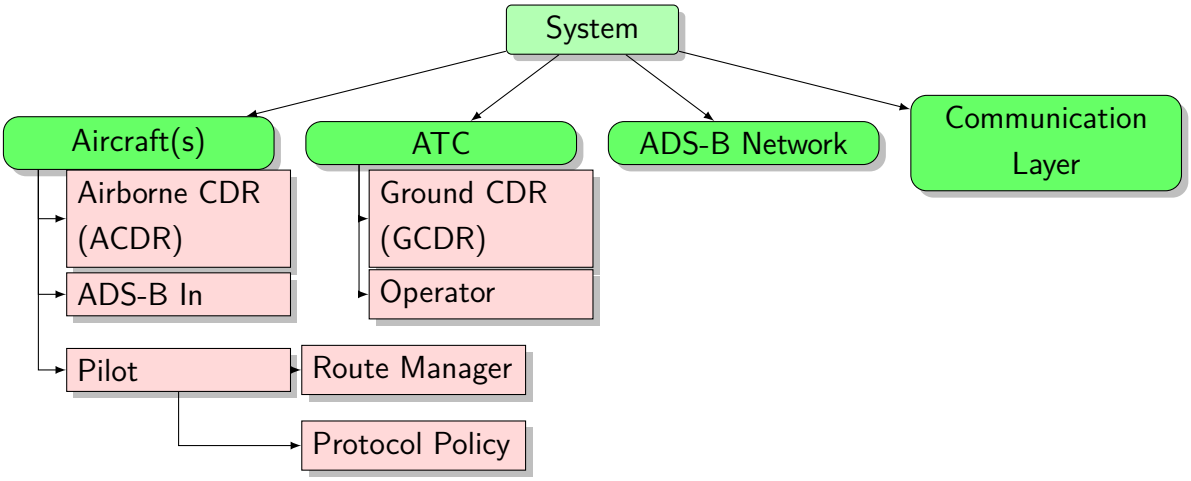


Figure 29.5: Hierarchical decomposition

all combinations of conflicts between aircraft of different types: GSEP-to-SSEP, GSEP-to-GSEP, SSEP-to-SSEP. This abstraction only represents how many aircraft can be in a single conflict at the same time, and does not assume anything on the size of the airspace.

Figure 29.5 shows the decomposition of the system into a hierarchy of component types. This provides an architecture that can be refined. For example, we break down the definition of the Aircraft and ATC components into sub-components. By breaking down the behavior of each component into its sub-parts, we are able to simplify modeling and validation.

We use the Aircraft component (the most complex component) to exemplify our parametric modeling approach. There are two types of aircraft: SSEP and GSEP. Since these two types differ only in few ways, they are modeled as a generic aircraft component with some additional parameters. The parameters capture some of the dimensions described in Table 29.1. An additional parameter is used to enable/disable the ADS-B receiver (ADS-B In), which is a feature restricted to SSEPs.

We model the *Aircraft* component as having the following parameters: `adsb_in`, `ts_agent`, `ss_agent`, and `burdening`. The parameters `ts_agent` and `ss_agent` are used to specify who is in charge of the Tactical Separation

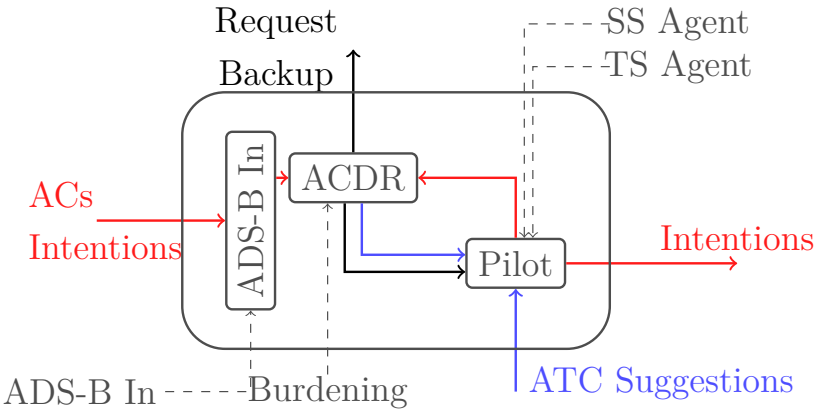


Figure 29.6: Aircraft Component

(TS) and Strategic Separation (SS). Similarly, the parameters `burdening` and `adsb_in` capture, respectively, the information about the burdening rule in use and the availability of the ADS-B receiver. Using this parametric model, we can describe a GSEP as an aircraft that is always separated by ground, and that does not have an ADS-B In component: `Aircraft(adsb_in=No, ts_agent=ATC, ss_agent=ATC, burdening=GSEP)`.

In each single configuration, we enforce that all GSEPs must have the same parameters; SSEPs are similarly restricted. Therefore, in the same configuration there cannot be two SSEPs with, e.g., two different separation assurance agents. This is not a limitation of the model or tool, but a design choice motivated by the domain that we are exploring and our choice to keep the model more understandable and limit the scope to realistic scenarios. The impact of the parameters in the component hierarchy is shown in Figure 29.6. Having components whose implementations are independent of the parameters makes it possible to *re-use* the components for multiple configurations. In Figure 29.6, we can see that the aircraft provides its real intention directly as output. The sharing of this information with other aircraft and the ATC is handled in the ADS-B Network and in the Communication Layer. By using dedicated components, it becomes

Type	**Name**	**Domain**
Parameter	id	[1..4]
	adsb_in	Boolean
	ts_sa_agent	{ATC, SELF, SATC}
	ss_sa_agent	{ATC, SELF, SATC}
	burdening	{Undefined, GSEP, SSEP}
Input	suggestion_{near,mid,far}_ground	Conflict Area [0..4]
	communication_phase	Boolean
	ac_{1,2,3,4}_intention_{current,near,mid,far}	Conflict Area [0..4]
	ac_{1,2,3,4}_{ts,sa}_agent	{ATC, SELF, SATC}
Output	intention_{current,near,mid,far}	Conflict Area [0..4]
	predicted_conflict_{near,mid,far}	Boolean
	request_{ts,ss}_sa_ground	Boolean

Figure 29.7: Parameters, Inputs and Outputs of the Aircraft model

easier to localize faults and experiment with different levels of information sharing, while keeping the output of the aircraft as ground truth. Figure 29.7 provides a summary of the input and output information, and of the parameters.

We capture the architecture described in Figures 29.4 and 29.5 using the OCRA language. OCRA also provides a means to write contracts in Linear Temporal Logic that describe the expected behavior of each component of the system. These are used to perform a first validation of the component implementation. To draw a parallel with software engineering, the contracts that we write are comparable to unit tests in which we focus on the correctness of the component in isolation.

Breaking components (e.g., Aircraft) into simpler components simplifies both modeling and validation. In particular, we can write properties about the aircraft and then decompose them into properties of the subcomponents. We do so by using contracts. For example, we write a contract for the aircraft (Figure 29.8) and decompose it into contracts on its

subcomponents.

```
1 CONTRACT AC_maintain_intention_ts_self
2 -- If self-separating, during communication phase if no conflict is
3 -- predicted, the intention will not change. Tactical Separation Case.
4 assume: TRUE;
5 guarantee: always ((communication_phase and ts_sa_agent = SA_SELF) implies (
6   (not predicted_conflict_near implies next(intention_near) = intention_near)
7    and
8   (not predicted_conflict_mid implies next(intention_mid) = intention_mid)));
9
10
11 CONTRACT AC_maintain_intention_ts_self
12 REFINEDBY cdr.ACDR_no_conflict_means_maintain_near,
13           cdr.ACDR_no_conflict_means_maintain_mid,
14           pilot.Pilot_apply_ts_self,
15           pilot.Pilot_intention_is_not_nop;
```

Figure 29.8: Example of a contract on the Aircraft component

To take advantage of contract-based design we need to perform two steps [59]. First, we need to check that the refinement of the contract is correct. This means that the guarantees provided by the subcomponents in the refinement are sufficient to prove the guarantee of the supercomponent. After performing this step, we know that independently of the choice of parameters, if the implementations of the ACDR and Pilot satisfy their contracts, then also the Aircraft satisfies its contract. As a second step, we verify that the implementations of each component satisfy their contracts. This operation is done locally on the component in isolation and, since most components are relatively small, it can be performed efficiently. Every time we modify a basic component, we only need to validate it against its contracts, and we are guaranteed that the composite components will still satisfy their contracts. This significantly speeds up the design loop.

An added benefit of this process of contract decomposition is the need to better understand the relation between the components. This raises interesting questions about how to define the components, how to divide responsibilities, and what behavior can be expected by every component in nominal situations. In fact, we are forced to define requirements that all components implementations must satisfy. In our case, this investigation

was supported by a close collaboration with the AFT group at NASA, and resulted, for example, in the definition of multiple possible ACDR implementations, and the definition of more than 130 contracts.

29.5 Configuration Analysis

Once the unified model is completed, we proceed to analyze each possible configuration in isolation. For each configuration we break the analysis into the following steps:

1. Instance Generation
2. Airspace, Nominal, and Extended Validation
3. Nominal and Extended Verification
4. Fault Tree and Reliability Analysis

Automation of this phase is very important. Each step is run automatically, from the definition of the instance to the generation of all verification and fault-tree artifacts. This ensures that the process is reproducible and scalable.

Instance Generation We associate each leaf component in our hierarchical architecture with an implementation (a behavioral model defined using an SMV file) by defining a *map* file. The OCRA tool uses this mapping to generate a monolithic implementation (SMV) of the instance. This makes it extremely easy to instantiate the system with multiple functional implementations of the components, and also to create instances with and without faults. We pass parameters through the OCRA architecture using pre-processing instructions to define constants. In this way, the variability

of the model is limited to the OCRA architecture and map files used during the generation phase. The outcomes of this phase are three models: *airspace*, *nominal*, and *extended*. These are standard SMV files, without parameters, that can be analyzed by any out-of-the-box technique.

Airspace, Nominal, and Extended Validation The models for the configuration are generated automatically. Before proceeding to the verification step, we need to gain confidence in the quality of the generated model. For this reason, we perform these additional steps of validation.

The scenarios that are taken into account in this work represent the interaction between a *controller* (the Air Traffic Control and the CD&R on-board), and a *controlled system* (the set of aircraft). In order to avoid a vacuous verification we first need to validate separately *controllers* and *system*. In particular, the *system* must allow the occurrence and resolution of LoS, and the *controllers* should should accept any possible trajectory intent from every aircraft. We generated these models by mapping the separation agents, or the aircraft, to implementations that have no constraints, while using nominal implementations for the other components. To certify that the components work correctly together, we verify 18 CTL properties encoding the possibility of bad and good behaviors, and 24 LTL properties derived from contracts.

The *nominal* model uses a nominal implementation for every component, including separation agents. Unlike the extended model, in this case we do not allow components to fail. We validate this model with 29 LTL properties derived from the contracts of the components.

Finally, the *extended* model uses an implementation for every component that includes faults, resulting into a total of 95. The validation of the extended model checks that all faults are possible (through 137 CTL possibility properties), and that they respect their dynamics, i.e., permanent

or transient, with 29 LTL properties.

Overall, the validation of the 3 models requires a combination of different techniques in order to be effective and be carried out in a limited time. The CTL verification requires a fixpoint-based approach, using BDDs, while for the LTL properties, we use the IC3-based algorithms implemented in nuXmv. Every property is checked against a known result, such as the negation of some existential LTL property, that, if violated, causes the analysis to stop for further investigation.

Nominal and Extended Verification In this step, we characterize different configurations by verifying additional properties. The most important is whether LoS can always be avoided (NO-LOS), followed by stronger versions: NO-LOS-Near, -Mid, -Far. Other properties provide additional information on the quality of the configuration, e.g., "Every conflict in the Near-window (Mid-, Far- respectively) is detected by at least one Agent." (Detect-Near, -Mid, -Far). This provides a simple way of ranking configurations for further investigation. During extended verification, instead, we check whether these properties are still satisfied in the presence of faults. For most properties this will not be the case. However, if some property is satisfied even with faults, it means that the property and the faults have no relation in the given configuration. In this step, we verify 24 LTL and 30 invariant properties on both the nominal and extended models.

Fault Tree and Reliability Analysis We compute the Fault Tree associated with each safety property in order to understand the resilience of each configuration to faults. We compute them automatically from the formal model, using the IC3-based technique for minimal cutsets computation described in Part II. For each Fault Tree, we also generate a *reliability* function as described in Section 20.3, which relates the probability of violating

the property to the probability of failure of each basic fault.

29.6 Data Analysis

Each configuration can be analyzed independently. We exploit this fact and run the analysis on a cluster with 12 Intel Xeon X5650 processors (72 cores). The average size of the models was 10^{107} states, and each model was checked against 346 properties. The two most difficult steps were those of model validation, due to the need for BDD-based reasoning, and minimal cutset computation, since it requires solving a parameter synthesis problem. These two steps were completed within an hour for most configurations, but for roughly 10% of the models, they required several hours to complete. Verification of the LTL properties was performed using the nuXmv IC3 implementation, requiring roughly 5 minutes per model.

Once all results are available, we can perform the last step of the process: *Data Analysis*. Each configuration provides us with a set of verification results and a set of fault-trees. These artifacts can be collected into relations. The first, $V \subseteq C \times \mathbb{B}^n$, relates each configuration to the satisfaction of the verification properties. The second, $FT \subseteq C \times \mathbb{N} \times 2^{MCS}$ instead relates each configuration and property to the set of minimal cutsets associated with it. This data can be queried and manipulated offline, by the domain experts, in order to obtain more insights on the design space.

29.6.1 Summary of Results

Most of the configurations (Figure 29.9) satisfy the key property of avoiding Loss of Separation (NO-LOS). The fact that NO-LOS-Far is satisfied by some SSEP-Only configurations is due to the non-receptive implementation of the ACDR, which assumes that trajectories are computed in a way that avoids potential conflicts in the Far window.

	GSEP-Only 4-0	**Mixed** 3-1	**Mixed** 2-2	**Mixed** 1-3	**SSEP-Only** 0-4	**Total**
NO-LOS	324	244	212	213	258	1251
NO-LOS-Near	324	244	209	210	252	1239
NO-LOS-Mid	324	192	138	141	198	993
NO-LOS-Far	0	0	0	18	84	102

Figure 29.9: Models satisfying NO-LOS for different windows

Prime Implicants To extract interesting facts from the verification results, we synthesize the region of parameters that satisfy a property of interest. This result has beed carried out by fixing the property value and quantify away the other properties in the relation V. E.g., for NO-LOS:

$$NO\_LOS(C) = \exists P_1, \cdots, P_n.\ V(C, P_1, \cdots, P_n) \wedge NO\_LOS$$

where P_i is a Boolean variable associated with the verification result for property i, and C is the set of configuration variables. In this way, we can compute the region of parameters associated with the satisfaction of each property. Very few of these regions have a compact representation. To extract interesting facts from these regions, we compute the *prime implicants* of the region, i.e., the set of minimal elements that are sufficient to enforce the satisfaction of the property. For cardinality 1, we obtain the following implicant for NO-LOS:

$$(MIX = \langle 4, 0 \rangle) \vee (SSEP\_TS\_SA = ATC) \vee (SSEP\_SS\_SA = ATC)$$

This tells us that there are two ways to guarantee NO-LOS: having only GSEP airplanes, or having the ATC in control of the Strategic or Tactical separation of any SSEP.

We also verified the claim from above, by checking that NO-LOS-Far is achieved only by configurations using non-receptive ACDR. Moreover, we verified that not all configurations using non-receptive ACDR can satisfy

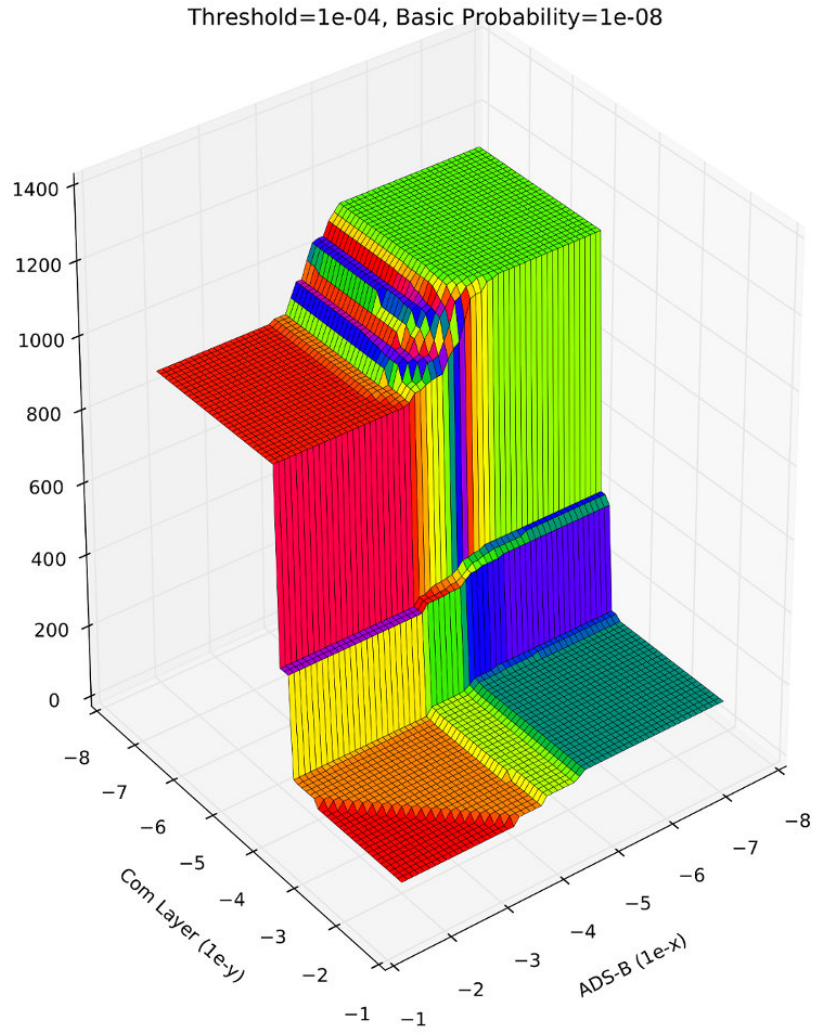


Figure 29.10: Impact of the communication faults on LOS probability.

NO-LOS-Far, thus discovering a necessary but not sufficient condition. These analyses were performed using pySMT [70] in order to represent the data using BDDs [47] for efficient querying.

Reliability Functions Analyzing the reliability functions, we can synthesize the region of configurations that have a probability of violating a property lower than a given threshold. This result provides us different sets of possible candidates that are able to guarantee a high reliability. In addition to

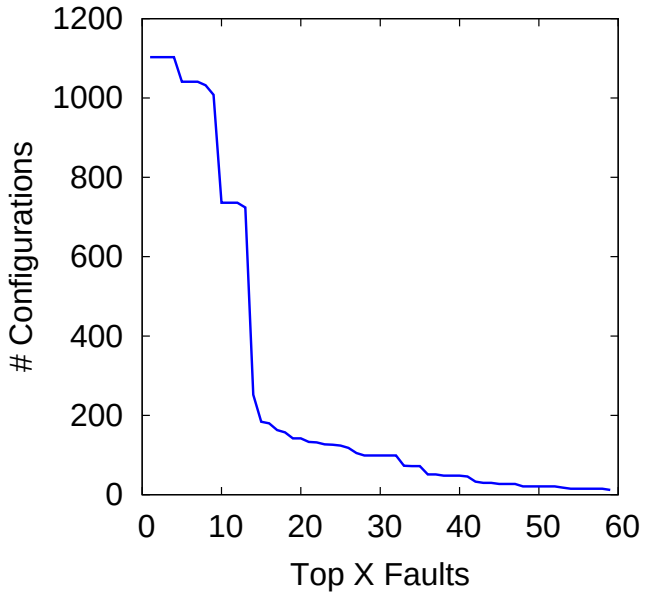


Figure 29.11: Configurations impacted by the top N Single Point of Failure.

that, we want also to analyze the impact of a variation in the probability of failure of different groups of components. In Figure 29.10, we demonstrate this last analysis by plotting how many configurations have a probability of leading to a LOS that is below the threshold of 10^{-4}, when changing the probability of the faults of the Communication Network (y axis) and of the ADS-B Network (x axis). A different analysis is presented in Figure 29.11 in which we analyze how many configurations share the same top N single points of failure (minimal cutsets of cardinality one). We can see that there are roughly 10 single points of failure that are shared by more than a thousand configurations. However, we also notice that most faults are single points of failure for a limited number of configurations; recall that there are 95 faults in total.

29.7 Detailed Comparison

The analysis shown in the previous section are tailored to extract common patterns and characteristics in different function allocations. The outcome

of that analysis may ask for a more detailed evaluation of a limited subset of the possible architectures. In this case, the comparison elaborates on the details of the minimal cutsets and reliability functions.

29.7.1 Minimal Cutsets Comparison

A common practice in Fault Tree Analysis consists of comparing the size of cutsets of the same cardinality. This approach is based on the intuition that the fewer the single point of failures in the system the higher is the overall reliability. This approach can be extended also to the cutsets of higher cardinality e.g., double failures. This approach provides an intuitive understanding of the relation between different fault trees, however, it is not always precise, since a single failure might be less probable than a double failure.

An example of this analysis is presented in Table 29.2, which compares the results of the FTA with LoS as TLE, by varying i) the number of GSEPs (G) and SSEPs (S), with SATC on strategic separation and ATC on tactical; and ii) the ability to share far intention from the GSEPs (E for Enabled, and D for Disabled) . In this example the number of single point of failures does not vary for every configurations (i.e., 5), while the number of double failures decreases when the GSEPs share their far intentions with SSEPs aircraft. Important fact, however, is that the number of triple failures increases when GSEP-far is enabled. This behavior in the fault tree analysis results is typical when adding redundant components. In fact the idea behind redundancy is to increase the fault tolerance, and essentially what is a single point of failure becomes a double (or higher) failure.

Further analysis on fault trees can be performed by evaluating the minimal cutsets that are not in common. An example of this analysis can be done by considering the configuration 2G-2S, and comparing the fault trees obtained with the TLE “there is a LoS between SSEP1 and GSEP1”,

Card.	3G-1S		2G-2S		1G-3S	
	E	D	E	D	E	D
1	5	5	5	5	5	5
2	12	15	12	16	12	15
3	33	24	35	23	36	27
...	...		...		...	

Table 29.2: MCS, LoS as TLE, and GSEP-far (E/D)

when varying GSEP-far.

The results of this evaluation shows that if GSEP-far is disabled then the fault configuration FC = {G1.$F_comm_$ATC$_tot$, S1.$F_comm_$ATC$_tot$} can cause the occurrence of the TLE. Differently, when GSEP-far is enabled, FC is no more a necessary condition to reach the TLE because the ACDR on the SSEP is able to react to that situation. In fact, if GSEP-far is enabled then FC requires to be combined respectively with {ATC.F_far_res},{ATC.F_future_res}, {G1.F_comm_adsb}, and {S1.$cdr.F\_future\_resolve$, S1.$cdr.F\_resolve\_detection$} to cause the occurrence of the TLE. Thus, the enabling of GSEP-far turned a minimal cutset of cardinality 2 into 3 cutsets of cardinality 3 and 1 of cardinality 4.

29.7.2 Reliability Function Evaluation

We formally analyze the set of possible AAC designs early in the system design phase, before specific module implementations or probabilities of failures are fully defined. However, we can evaluate how the reliability functions compare to each other by analyzing different possible probability values. For instance, if we take into account the probability of reaching a LoS between two aircraft of the same type (for instance GSEP1/2 and SSEP1/2 in the scenario 2G-2S), then we expect that the failure of the ATC will affect more the GSEPs than the SSEPs. This can be assumed

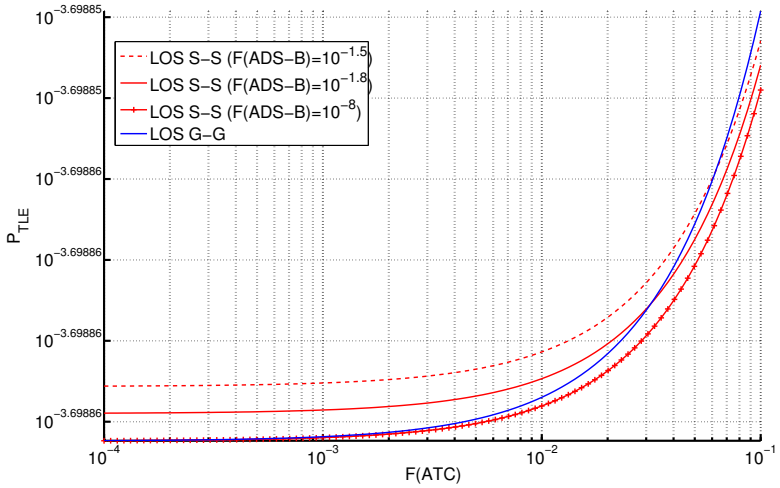


Figure 29.12: Reliability comparison between different aircraft types

considering that SSEPs aircraft rely on ATC for strategic separation only as a backup, while they are self-separating otherwise. However, the ACDR onboard of the SSEPs highly depends on the ADS-B system and its possible failure. Fig. 29.12 shows the result when varying the probability of failure of the ATC (x-axes) and the ADS-B (red lines), by keeping fixed all the other values. According to the results, there exists a probability of ADS-B failure such that the pure ATC-based separation assurance between two GSEPs (blue line) is more reliable than the one implemented by the SSEPs aircraft.

We need to remark that the aim of this evaluation is to provide the functions that relate probability of TLE occurrence to the probability of failures of each component, and not the actual values of failure probability. In fact, the outcome of the reliability evaluation is a set of functions in Matlab format that can be analyzed using common analytical numerical tools. Thus, the remarkable aspect of such type of artifacts is that they do not need to be recomputed when the real component implementation will be defined.

29.8 Interesting Executions

A selection of the most relevant results was discussed with the domain experts. In particular, we were able to independently reproduce two known issues, side-walk [120, 79] and coincidental conflicts [74], and discover a new one.

Side-walk Conflict. Side-walk conflicts occur whenever we use the "simple" implementation of the ACDR, in which conflicts between SSEPs are resolved by choosing a free conflict area. The problem occurs when more than one SSEP decides to move to the same conflict area. Due to the symmetry of the resolution algorithm, this strategy is not guaranteed to resolve the conflict. To break this symmetry, we developed the asymmetric version of the ACDR.

Coincidental Conflict The asymmetric ACDR is not able to resolve conflicts early. In particular, we would like to always satisfy NO-LOS-Mid, i.e., avoid predicted LOS in the Mid window. This is not possible if we allow only one communication step. In fact, if four aircraft are in two different conflicts that are resolved correctly, they might still end up in a new conflict. Consider the two conflict sets: {AC1, AC2} and {AC3, AC4}. AC1 and AC3 decide to move to solve their conflict. However, they chose to move to the same conflict area. An additional round of communication is needed in order to resolve this conflict.

Backup From Ground. The novel problematic configuration that we identified stems from limited requirements on the behavior of the backup operation, i.e., when an SSEP is able to request backup from ground and it delegates its separation to the ATC (SATC). This turned out to require more assumptions than were initially considered. In fact, when enabling this behavior, all configurations violate NO-LOS, excluding the ones with non-receptive ACDR. This is motivated (as shown by the counterexamples)

by a lack of information and a mismatch of expectations in the airspace. In particular, in the design used in this project, whenever an aircraft requests ATC assistance, the other aircraft are not aware of it. Therefore, all of the other SSEPs expect the aircraft to maintain its behavior as an SSEP. In order to solve this issue, we propose two options. First, requests for ground-assistance are relayed to other aircraft. Second, the algorithm for separation used by ATC needs to take into account that the aircraft was an SSEP, and therefore compute a resolution taking into account what the other SSEPs expect the aircraft to do. These extensions are left as future work.

29.9 Related Work

Before NASA turned to the question of what designs are best for automated air traffic control, it was necessary to explore what designs are *possible*. To that end, NASA launched several initiatives to formally reason about a *single* such system; two of these works, using symbolic model checking [124] and probabilistic model checking [125] techniques led to the decision to use the former for the problem of broader design space exploration.

In this work, we firstly designed and verified a monolithic model of the now-more-detailed design, then we generalize the analysis to deal with more than 1,600 configurations, and present a tailored process that allows us to model, validate, verify, and compare such a huge design space.

The term *design space exploration* is commonly used to describe the study of a design space (mostly combinatorial) by avoiding the computation of all solutions and optimizing with respect to some cost function. For example, Airbus [17] also uses automated techniques to evaluate design spaces. In particular, multiple solutions are compared and sorted with respect to their weight. It is important to notice, however, that we are dealing

with a sequential problem while works such as [17] deal with combinational ones. Moreover, the existence of a cost function allows the optimization engine to prune "bad" configurations, thus reducing the actual number of configurations that will be eventually checked. In our case, there is no cost function defined; we are instead interested in a better understanding of the design space, and thus want to be able to thoroughly analyze every possible design. Therefore, we analyze all of the realistic configurations and collect the data in a form suitable for subsequent comparison.

When we move from combinational to sequential problems, we find works related to product lines, e.g., Software Product Lines [61], that deal with a similar problem of verification of a parametric system. In [61] the authors propose an extension to NuSMV that is able to perform symbolic model checking of an extended version of CTL (feature-oriented CTL). The differences with our work are several. From a process point of view, we focus not only on the verification but also on the validation of the generated models and on safety assessment; the outcome of our process is more informative since it relates the set of configurations with the properties that are satisfied (i.e., parameter synthesis). Finally, we integrate the modeling phase with a compositional approach that helps to save significant modeling effort. In principle, we could try to combine multiple configurations in order to analyze them together in a symbolic way. However, for our case-study this was not needed. On the contrary, the ability to work on each configuration independently made it possible to exploit high levels of parallelism provided by modern HPC infrastructures.

29.10 Conclusions and Future Work

In this Chapter, we presented a case study on the application of formal methods to the analysis of the big design space associated with the Next-

Generation Automated Air Traffic Control System under study at NASA.

We combined existing techniques and tools to perform model generation, validation, verification, and safety assessment. We achieved this by using a compositional, parametric, and contract-based approach in order to maximise reuse, and to ensure great confidence in the models by means of aggressive model validation. To the best of our knowledge, this is the first time that a design space of this scale has been mapped out by considering every possible solution in such depth. Our approach resulted in a wealth of interesting data that supported the re-discovery of known facts, and also the detection of new insights. We provided some intuition on how to extract meaningful information from this data, but we expect that even more will be extracted in the future, working in collaboration with the NASA domain experts. This introduces interesting challenges for data analysis that could be explored in the future.

In the future, we also plan to extend the model by identifying additional modeling dimensions of interest, e.g., the fact that ADS-B information might not propagate equally to all aircraft, or the presence of multiple ATCs. We also plan to leverage more the contract-based infrastructure defined in this work, in order to identify properties that can be proved by pure compositional reasoning.

30

Reliability Analysis on Fly-by-Wire Architectures

Stability and control of an aircraft is subject to the ability of changing the angle of rotation in the three dimensional space. The Flight control surfaces allow the pilot to control those aspects and changing aircraft flight attitude, hence by operating on roll, yaw, and pitch angles. Pilot's controls (yoke, sticks, rudder pedals, etc.) drive the flight control surfaces, and in a mechanical or hydro-mechanical system such interaction relies on pulleys, cranks, tension cables and hydraulic pipes. This approach compares to the more advance Fly-by-Wire (FBW), which translates the pilot's commands into electric signals, transmits them by wires (hence the fly-by-wire term) to flight computers that determine how to move the actuators at each control surface to provide the ordered response. Both mechanical and FBW systems have advantages and disadvantages, however current direction in civil aircraft demands for the more reliable, lightweight, and autonomous electrical system.

Guarantee reliability, safety, and availability is most assuredly a crucial part when developing flight control systems, and loosing aircraft maneuverability should be a very unlikely event. Recent techniques, such as the one presented in Part IV, allow us to analyze architecture and different

approaches to FBW systems. This analysis can be done in the early stage of the design phase, thus we are able to analyze the potential capability of the different approaches, without taking into account the concrete implementations of each single block.

The advantages that a Fly-by-Wire approach offers over the traditional mechanical flight control systems are dramatically important in modern civil aircraft. The FBW system replaces heavy mechanical control systems with lightweight electrical wires. First of all, this guarantees a lower fuel consumption, in addition to a simplified system maintenance. Moreover, a FBW approach turns out to be highly compact, and the gained space can be used to increase the passenger capacity.

In a pure hydraulic system, the pilot has full control of the aircraft, and any maneuver of the flight control surfaces is subjected to a pilot command. However, in a FBW configuration the pilot provides the inputs to the flight computers, which drive the flight surfaces to serve the request. A significant difference of those two approaches can be understand with a simple example. Consider an aircraft that is flying at a constant speed with pitch and roll angles equals to 0. In this case, when the pilot is not providing any command to the system, the mechanical flight control will interpret it as "no changes to flight surfaces", while the FBW will just preserve the flight attitude.

Relying on a FBW system allows engineers to design an aircraft with relaxed stability i.e. the tendency to change flight attitude, and control it via automated electrical systems. This allows for an optimization of flight efficiency by reducing drag (air resistance), and increasing maneuverability. This results into a tight dependency between FBW system and mechanical development of the aircraft.

The first civil installation of an electrical based flight control was the one designed by the Aerospatiale to cover the needs of the Concorde in 1969,

and was based on analogical computation. The Airbus A310 program in 1983 introduced the first version of a digital electric Fly-by-Wire system. Lather advances of the Airbus A340 improved the previous technologies, and in 1992 it defined the base for the modern FBW systems, where all flight control surfaces are controlled by digital electronics. The Boeing company integrated a FBW approach with the 777 family in 1994.

Airbus and Boeing commercial airplanes differ in their approaches and principles in using fly-by-wire systems, considering both physical implementation an philosophy. Airbus aircraft do not permit pilots to operate over some predefined limits of maneuverability. With the Boeing 777 model, the pilots always have the possibility to override the flight computer, and may operate the aircraft beyond its usual range of maneuvers under standard situations. Modern aircraft no longer have a mechanical backup, thus the reliability of FBW systems is crucial to guarantee safety.

In this Chapter we present the two systems and analyze the their reliability aspects. More specifically, Section 30.1 describes the Fly-by-Wire principles and main characteristics. The FBW system of Boeing 777 and Airbus A340 are analyzed respectively in sections 30.3 and 30.4. To conclude, section 30.6 provides the future directions of this work.

The system modeling of the case studies presented in Sections 30.3 and 30.4 come purely from the inspection of publicly available documentations, thus they may differ from the real implementations. The purpose of these case studies are purely directed to show how the techniques in Part IV can be applied on a real-world redundant architecture.

30.1 Fly-by-Wire Principles

The term Fly-by-Wire describes a control system that relies on electrical signals, rather than mechanical ones, to translate pilot's command into flight surfaces movement. The basic approach to a FBW system consists in substituting the conventional mechanical pilot's controls with electrical input devices. Those devices translate their position into an electrical signal, which is sent to the actuator electronics that impose an angular displacement of the control surfaces. Those angles are proportional to the input, without any form of enhancement. Possible forces that bind actuator movements are provided as feedback to pilot's input devices. This simple approach is the one introduced in the 1970s with the Concorde program. Later designs take advantage of the Fly-by-Wire potential by integrating stability augmentations, and flight envelope limiting[1] [67]. In this case, the position of control surfaces is no longer a direct representation of pilot's command, as well as actuator movements feedbacks, which are not directly provided to input devices.

The high-level operations of a Fly-by-Wire system can be represented with the block diagram shown in Figure 30.1. The Pilot operates on cockpits controls, and provides the *Objective* to the system as electrical signals. Those signals are provided to the FBW system (assume that *Objective* and *Revised Objective* are equal in the first iteration), which computes the *Command* for the *Control actuators* system. The *Computed Order* is then provided by the *Control Actuator* to the *Control Surfaces*, which actually impose an *Aircraft Maneuver*. A set of *Sensors* read the actual state of the *Aircraft*, and those signals are then sent back to the FBW system. The *Aircraft* commands are revised by the *Command Computation* according with the difference between *Command* and *Actual State* of the *Aircraft*.

[1] Automated protection against exceeding predefined flight parameters like speed and yaw, roll, and pitch angles is called *flight envelope protection*.

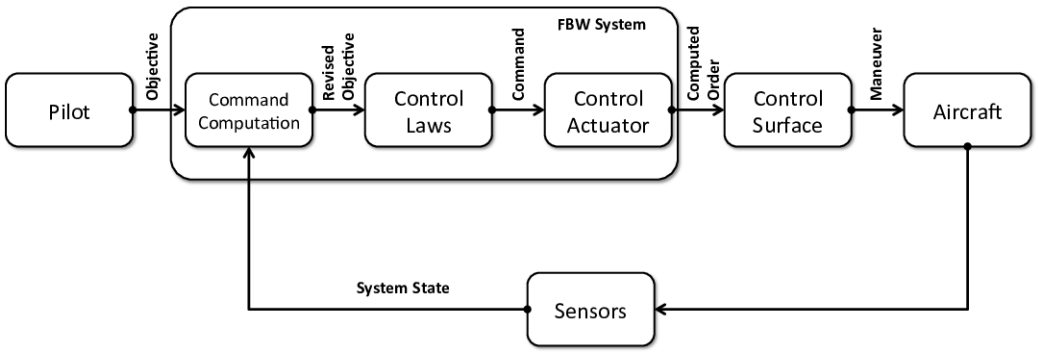


Figure 30.1: Standard Fly-by-Wire loop

The FBW architectures that we analyze in this Chapter consider *Command Computation* and *Control Laws* components, while *System State* is incorporated in the inputs of the FBW, thus they are not explicitly considered.

30.2 Formal Analysis Process

In this Chapter we apply the techniques for architecture reliability analysis described in Part IV, on two different Fly-by-Wire architectures. The formal process that we applied is tailored to analyze redundant and safety critical systems. This Section provides an overview of such methodology.

Redundant and Safety Critical System As described earlier in this Chapter, modern aircraft no longer have mechanical backups for aircraft piloting. Due to this fact, a malfunction of the Fly-by-Wire system during flight operations is not an option. However, any electrical or mechanical part has a possibility, even extremely remote, to fail. In fact, a redundant and safety critical system like a Fly-by-Wire has to be designed in order to be functional and available even in presence of multiple components failure. This characteristic opens up to a variety of operational modes, that range

from *Normal*, where no failures are detected, to different levels of *Degraded* configurations. If the system is in a *Degraded* mode operational level, then the overall system behavior should be preserved even though not all components are active and functional. In fact, systems like the redundant Fly-by-Wire are designed in order to detect failing components and disable them. In fact, a subset of the degraded modes is usually able to guarantee the minimum level of safety requirements.

Modeling Techniques The modeling approach that we applied on the Fly-by-Wire systems relies on the OCRA/xSAP tool-set, as described in Part V. This approach requires modeling the system architectural description, and integrating it with a mapping that links each leaf component to an implementation coming from a library of SMV files. Each implementation file is modeled with SMV language, and describes a common component implementation such as triple modular redundancy, computational module, median value selection, communication bus, and so on. The possibility of relying on a library-based implementation allows the modeler to dramatically reduce the modeling time, while still having enough expressivity to describe a real-world redundant architecture.

Faults Definition and Categorization In order to evaluate the reliability of a redundant system, the formal modeling also has to consider failure events. In this evaluation we analyzed the impact of two types of faults: visible and hidden. A computational component, with input and output ports, that is experiencing a visible fault will set the outputs to a specific value (for example 0.0) that express that the signal is invalid. The occurrence of a hidden failure enables the possibility to provide a wrong output, but not distinguishable from the correct one. In other words, the first case gives an additional information on the validity of the output signal, condition

that does not hold for the latter.

The information regarding the validity of the signal can be used by non failing components to exclude invalid values from the computation. An example of this approach is applied to the voter mechanism, where the invalid signals are simply not considered. In fact, this adaptation of the implementation is able to mask 2 visible or 1 hidden failure. In practice, parity checks, cyclic redundancy checks (CRC), and hash codes are some of the possible techniques that can be applied to recognize hidden failures, and thus modify them into visible ones.

Given the probability of occurrence in one hour of operation, for each fault event, the Fault Tree Analysis allows us to estimate the probability of the Top Level Event. In the following case studies we set the occurrence probability of a visible failure to 10^{-7}, while for hidden failures the value is set to 10^{-10}. Those values represent an "extremely improbable" and an "extremely remote" event, whose categorization is standardized by the authorities Europe and the United States [9, 41]. Table 30.1 describes the complete probability events classification, in addition to the color convention that we apply on the rest of this Chapter.

Abstract Model Analysis Model validation is a fundamental part in model based formal analysis. In addition to standard property verification, in this work we analyzed also an abstract representation of the concrete system, by relying on predicate abstraction techniques. In particular, the nuXmv model checker exposes this functionality that consists in generating an abstract symbolic state machine given a set of (Boolean) predicates. From that representation it is possible to obtain its explicit version, where each state represents a complete reachable assignment to the predicates. The outcome of this analysis is visually represented, and thus compared with the expected result.

Probability	1	10^{-5}	10^{-7}	10^{-9}	0
Color	Red	Orange	Yellow	Green	Blue
Classification	Probable	Remote	Extremely Remote	Extremely Improbable	Extremely Improbable (No single points of failure)

Table 30.1: Probability Events Classification [9]

Fault Tree Analysis The evaluation of the systems under analysis considers two main sets of properties. The first set checks, for each operational mode, under which conditions the output of the system is not correct. Those requirements are formalized in the form $init\_mode \rightarrow correct\_output$, for any possible normal or degraded modes.

The second set of properties is intended to verify how the system changes the operational mode. This set of property is formalized as $init\_mode \rightarrow \neg final\_mode$, where init and final mode range over all possible modes.

30.3 Boeing 777 Primary Flight Computer

30.3.1 System Description

The Fly-by-Wire system of the Boeing 777 extends the standard triple modular redundancy approach, by implementing a triple-triple redundancy [122, 96, 81, 110, 114]. The Primary Flight Computer of the Boeing 777 is composed of 9 identical Lanes, where each of them has full flight

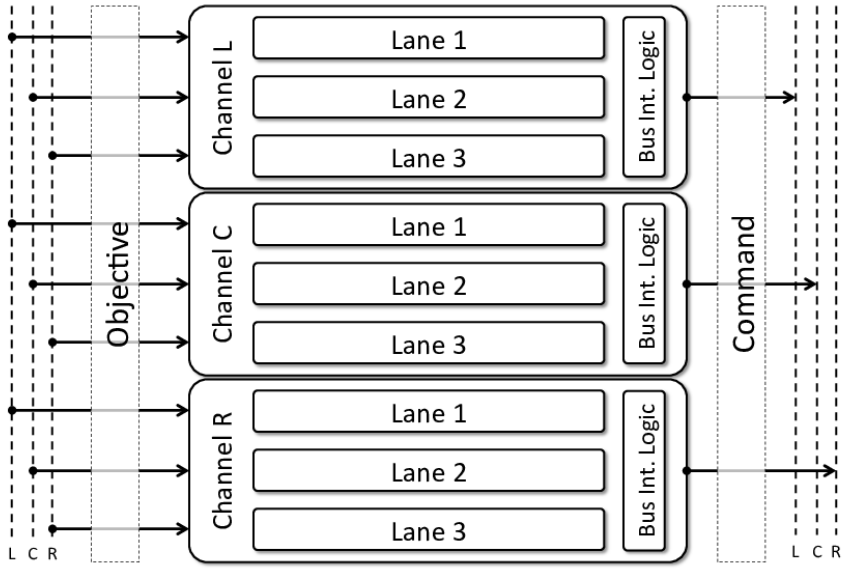


Figure 30.2: Boeing 777 FBW Architecture [122]

control capability of translating pilot's *objective* into actuators *command*. The lanes, as shown in the architectural description in Figure 30.2, are grouped by 3 into Left, Center, or Right Channel. Each channel communicates with the others via a triple Bus, namely Left, Center, and Right Bus. The *Bus Interface Logic* is an artificial component, not present in the original design, that implements the bus communication protocol.

The redundancy aspects of the PFC allow the system to be fully functional, even if some of the lanes are not behaving correctly. This capability requires to recognize whether a lane cannot be considered trustworthy in the command computation, so it needs to be disabled. This function is implemented in the Boeing 777 PFC where one lane for each channel computes the actual commands, and the remaining two lanes that monitor the correctness of the computed command. This requires each lane to be able to operate in two different modes: *Command* and *Monitor*.

Power System

The PFC requires electrical power to work properly, and this results in a tight dependency between those systems. Thus, the power system also has to be designed to be fault tolerant and reliable.

The main source of power in a modern aircraft is provided by the engines, and the Permanent Magnet Generators (PMG) components transform kinetic energy into electrical power. The Boeing 777 relies on a two PMGs per engine, with a total of four PMGs (i.e., it is a twin engine aircraft). A single PMG is sufficient for the energy provisioning of the entire aircraft, however a double engine failure would disable all electrical generators. As described earlier in this Chapter, modern aircraft have no mechanical backup and the loss of electrical power would result in a catastrophic condition. In order to mitigate those event, the power system relies on multiple batteries that provision the aircraft when no generators are available. The definition of the interconnections between generators and batteries is one of the challenges when defining a power system. Figure 30.3 shows the power system architecture implemented in the Boeing 777, which is composed of 4 PMGs, 3 batteries, and 3 powered buses. The power system is split into 3 disjoint sub-systems that provide electrical power respectively to left, center, and right PFC channels. The each battery has its own charger that acquires the power from the PMGs, however this aspect is not considered in this work but we modeled the batteries as alternative sources of electricity.

Channels Behavior in Normal Mode

The command computation of the Boeing 777 PFC is characterized by the sequential computation of four different signals:

1. Proposed Command Output (PCO): each lane first proposes an actu-

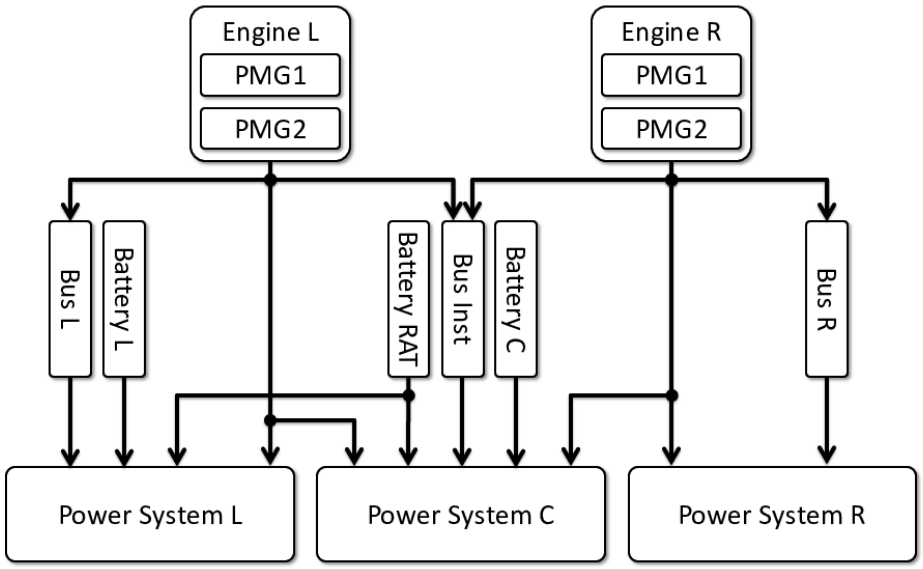


Figure 30.3: Boeing 777 Power System [114]

ators command;

2. Select Command Output (SCO): the command that is selected after reading all PCOs;

3. Cross Lane Enable (XLE): a signal that judges the sanity of the computed command. This is a lane to lane signal, local to a channel;

4. Cross Channel Inhibit (XCI): similar to the XLE, evaluates the sanity of the computed command. This is a channel to channel signal.

Figure 30.4 shows in detail the functional parts composing a Channel, when it is operating in *Normal* mode i.e., when no failures are detected. In this situation, the lanes are called *Command*, *Standby*, and *Monitor* lane, where the first one is operating in *Command* mode and the other two in *Monitor* mode.

Command Lane The *ISM CLAWS* component is the one representing the computational part, in fact it translates the pilot's objective into actua-

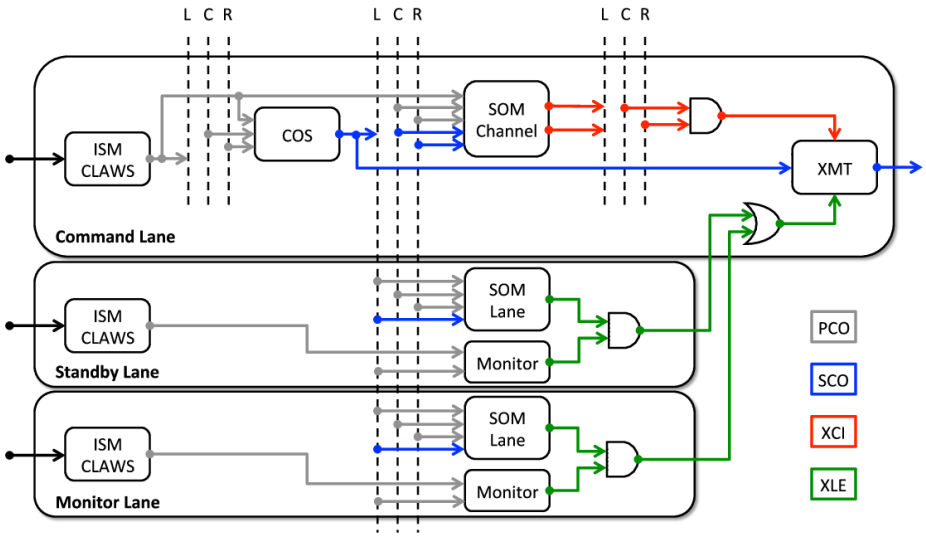


Figure 30.4: Boeing 777 Left Channel (Normal Mode)

tors command. The value computed by the *ISM CLAWS* at this stage it is called *Proposed Command Output* (PCO) and each channel writes it on its reference bus i.e., left, center, or right for respectively left, center, and right channel. Successively, the *Channel Output Selector* (COS) performs a median value selection (MVS) over the local PCO and the two PCOs computed by the other two channels. The result is called *Selected Command Output* (SCO), and it represents the final computed command. The next part of the computation is devoted to evaluate the behavior of the other channels, and send them a *Cross-Channel Inhibit* (XCI) when a wrong behavior is detected. In particular, this operation is performed by the *Selected Output Monitor* (SOM) component, which analyzes all PCOs and the SCOs coming from the other two channels. At last, if the AND of the XCIs coming from the other channels is False and the OR of the XLEs is True, then the SCO is written on the bus by the *XMT* component.

Standby and Monitor Lanes As shown in Figure 30.4, the operations in Normal mode of Standby and Monitor lanes are identical, and are directed

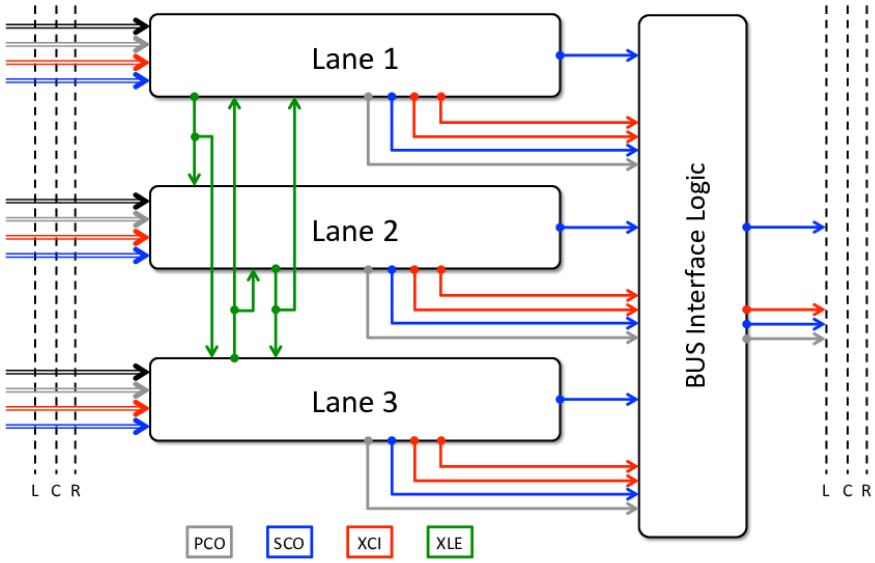


Figure 30.5: Boeing 777 Left Channel

to compute the *Cross-Lane Enable* (XLE) signal. More specifically, the *ISM CLAWS* components operates as in the Command lane, and provides a local PCO signal that is compared, by the *Monitor* component, with the Command lane PCO of the same Channel. The output of the *Monitor* is a Boolean value that is combined, in an AND logic gate, with the output of the *Selected Output Monitor* (SOM). This component reads the SCO from the Left Bus of the Command lane, and evaluates possible discrepancy by comparing it with Left, Center, and Right Channels PCOs.

Lanes Reconfiguration

Figure 30.4 shows the lanes behavior under nominal condition i.e., when all components are enabled. However, XCIs and XLEs signals are intended to detect lane malfunctioning, and exclude them from the command computation. As depicted in Figure 30.4, the command lanes are the ones that receive those disabling signals, thus are the only ones that can be disabled. In normal condition the role assigned is Command-Standby-Monitor, that

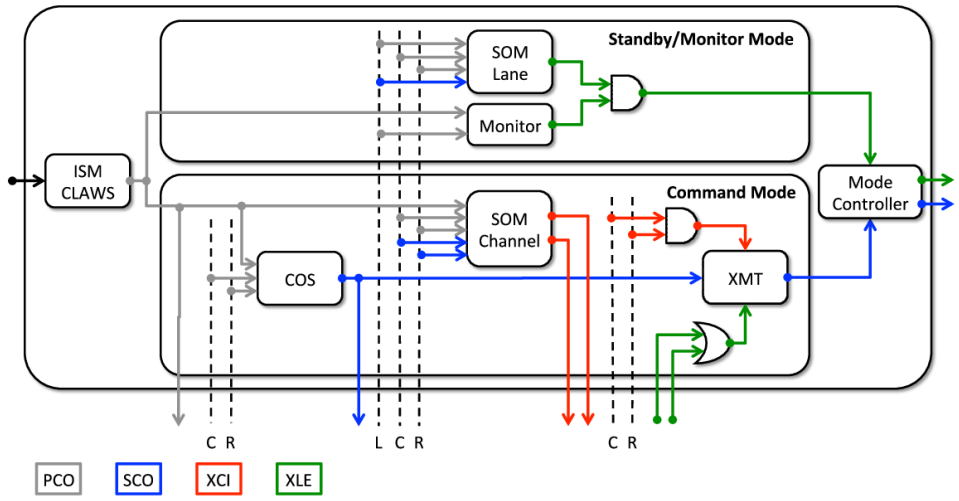


Figure 30.6: Boeing 777 Lane (of the Left Channel)

can change to Disabled-Command-Monitor when the first lane get disabled. Those partial availability conditions are called *Degraded*. The minimum number of active lanes in a channel is two, which means that additional lane failures will result in disabling the entire channel. The set of all possible degraded modes is listed in Table 30.2. More specifically, the situation when all 9 lanes are operational is called *Normal*, while *Degraded* identifies a mode where at least one lane is disabled. Each *Degraded* mode can be identified with the remaining *Full* (F) or *Partial* (P) operational Channels. A Channel is considered Partially available when 1 lane is not operational, and not available when 2 or more lanes are disabled. Table 30.2 lists all possible modes, by showing: i) mode name, ii) number of enabled lanes, iii) number of enabled channels, and iv) number of enabled lanes for each channel as an array of configuration in a canonical form (the values order is not relevant i.e., 332 represents also 323 and 233).

As described early in this Section, all lanes of the Boeing 777 PFC are identical, though they should have both Command and Monitor capabilities. A possible conceptual representation (of the left channel) that enabled

Mode	Lanes	Channels	Configuration
Normal	9/9	3/3	333
Degraded-2F1P	8/9	3/3	332
Degraded-1F2P	7/9	3/3	322
Degraded-3P	6/9	3/3	222
Degraded-2F	6/9	2/3	330
Degraded-1F1P	5/9	2/3	320
Degraded-2P	4/9	2/3	220
Degraded-1F	3/9	1/3	300
Degraded-1P	2/9	1/3	200
Dead	0/9	0/3	000

Table 30.2: Boeing 777 FBW modes

for lane reconfiguration is shown in Figure 30.5. In this representation, the output PCO, SCO, and XCI, previously written on the bus, are provided to the *Bus Interface Logic*. This component is the one that routes those signals on the buses, according to the current lane roles. With this representation, the XLE signals are simply fully connected between the lanes.

Internally, each lane has to integrate both command and monitor modes, in addition to having a "write-only" interface to the bus, as described in Figure 30.5. The implementation of a lane that obeys those requirements is depicted in Figure 30.6. In this representation, the ISM CLAWS component is shared between the two operational modes, and an additional mode controller has been devoted to disable the entire lane when requested.

30.3.2 System Faults Definition

As listed in Table 30.3, the resulting formal model of the Boeing 777 PFC has a total of 228 possible faults, where each of them is assigned to a probability of occurrence. In case of a visible failure, the probability is set

Component	# Failures	
	Visible	**Hidden**
Power System	15	0
Computational Modules	15	9
Common Cause Implementation	0	3
Channels	81	81
Buses	12	12
Total	123	105

Table 30.3: Components Failures

to 10^{-7} for one hour of operation, while for hidden failures the value is set to 10^{-10}.

The design of the Boeing 777 PFC aims to overcome also common implementation failures. In fact, if the 9 computational modules i.e., ISMs and CLAWSs, were sharing the same implementation, then a software or hardware bug would cause a loss of the entire system. The designers of the PFC approached this by relying on 3 different copies of compiler/processor, and distributing them to the channels with a rotation pattern i.e., ABC for Left, BCA for Center, and CAB for Right. Considering the software nature of these common cause events, their probability of failures cannot be quantified, thus it is set to 0.0. Even though the software implementation events are not taking part in the probability computation, they will be still considered and analyzed in the minimal cutsets computation.

30.3.3 Formal Modeling and Model Validation

The architecture description of the Boeing 777 PFC is modeled by relying on the techniques described in Chapter 20, thus the system is combinatorial. More specifically, in absence of faults the outputs of the PFC are a function of the inputs, and not dependent on the previous inputs or com-

puted values. In fact, under this assumption the whole PFC is equivalent to a single "ISM CLAWS" module. The extension with failures adds a non deterministic behavior to the system, however the PFC's outputs remain dependent only on current inputs and state variables. A combinatorial system gives the possibility to rely on simpler, and usually faster, verification algorithms due to the fact that the system diameter is known upfront.

The Boeing 777 system modeling required a total amount of 20 man-hours[2]. The resulting OCRA model is composed of 653 lines of code, that turned out to generate a miter construction in SMV language of 826 lines. The state machine generated has 763 state variables (277 real and 486 Boolean) where 228 of those are fault variables, and 9 uninterpreted functions.

The model validation of this system aims at analyzing the possible behaviors of the system, and to check that the reachable states are only the expected ones. For instance, in this phase we check that all modes listed in Table 30.2 are reachable, and that it is not possible to reach other system configurations. In addition to system level properties, we also formally verified that each leaf component implementation behaves as expected. This set of properties, not specific for the Boeing 777 PFC, integrate the SMV implementation library. This analysis resulted in the definition of a set of 626 invariant properties.

30.3.4 Abstract Model Analysis

The Abstract Model Analysis is a technique that allows the designer to inspect the behavior of the model. Applying the Abstract Model Analysis on a combinatorial system would result in a collection of initial states, thus providing a poor level of information. In order to overcome this issue, we extended the formal model by allowing the possibility to bound to one the

[2]This does not consider the time needed to finalize the system design.

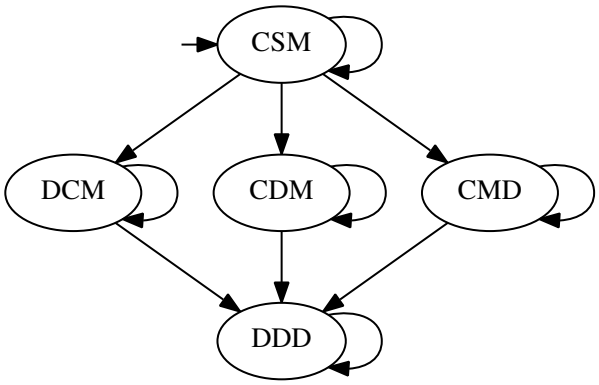


Figure 30.7: Boeing 777 Lanes Modes

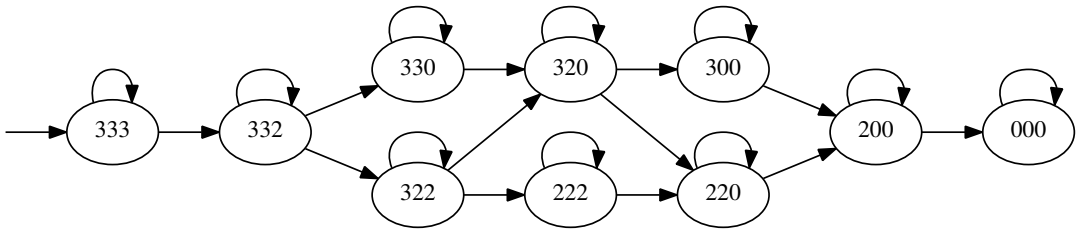


Figure 30.8: Boeing 777 Channels Modes (w.r.t, Table 30.2)

number of component failures for each unrolling of the transition relation. This extension maintains the set of reachable states, while it allows us to reach a much better representation in terms of human understanding.

An interesting set of predicates for the PFC system is the one that evaluates the operational mode of the channel lanes. In fact, each of the three lanes can be set as Command (C), Standby (S), or Monitor (M), and when one of them gets Disabled (D) the other two are assigned to C and M mode. When an additional lane is disabled then the whole channel is turned off, thus it gets into a DDD states. Figure 30.7 shows the result of this analysis. If no failures are detected (e.g., the initial state), the state of the system is CSM, which can either remain CSM or change into one of the degraded modes. No modes with two disabled lanes are reachable, therefore the failure of an additional lane will end to the DDD state.

After analyzing the lane modes at the channel level, we extracted the

abstract state machine representing the channels mode evolution. The list of expected configurations is reported in Table 30.2, where 3,2, and 0 represent respectively the lane modes CSM, DCM/CDM/CMD, and DDD. The abstract model representing the channels modes is shown in Figure 30.8. This result, as in the lanes case, shows a monotonic degradation of the active lanes, and it matches the expected reachable configurations listed in Table 30.2.

30.3.5 Fault Tree Analysis

The main aspect that has to be analyzed in a Primary Flight Computer is its robustness in providing the correct command to the actuators. In order to evaluate this aspect, we rely on the automated miter construction and evaluate under which fault configurations the system under analysis provides a wrong output. This analysis is performed on all possible initial configurations of the system i.e., from 333 to 000. Table 30.4 lists the results for this analysis, by showing the number of minimal cutsets for the cardinalities 0, 1, 2, and 3, with the resulting probability. The bold values express the number of fault configurations containing common cause implementation failures. We remark that if the analysis returns TRUE it means that the property does not hold even in the nominal case i.e., without triggering any faults. Clearly, this is the case when all channels are disabled (configuration 000).

The results in Table 30.4 allow us to evaluate if the system meets the functional availability requirements. In fact, [122] defines that the probability of providing a wrong output must be in the order of 10^{-10} when:

A. all lanes are operative; condition 333.

B. any single lane is inoperative in one, two, or all three channels; conditions 332, 322, and 222.

Initial Configuration	**Minimal Cutsets** 0	1	2	3	**Probability**
333	0	0	**9** 40	**257** 1309	$1.00 * 10^{-16}$
332	0	0	**33** 101	**2503** 4416	$3.51 * 10^{-16}$
322	0	0	**183** 128	**99** 7432	$5.31 * 10^{-16}$
222	0	**3** 0	128	8612	$5.32 * 10^{-16}$
330	0	3	**15** 300	**2288** 2751	$3.00 * 10^{-10}$
320	0	6	**180** 863	**21** 3442	$6.01 * 10^{-10}$
220	0	**3** 6	1163	1678	$6.04 * 10^{-10}$
300	0	31	**93** 145	**3** 160	$7.02 * 10^{-07}$
200	0	**3** 64	37	82	$3.40 * 10^{-06}$
000	TRUE				1.00

Table 30.4: Boeing 777 FBW: wrong output value (bold numbers are MCS with common cause failures)

C. any one channel is inoperative; condition 330.

D. any one channel is inoperative, in combination with any single lane inoperative in either one or both of the remaining channels; 320, and 220.

As shown in Table 30.4, the probability of obtaining a wrong output is higher than 10^{-7} when only one channel is available (orange cells). In all the other configurations the probabilities are in the order of 10^{-10} or lower, proving that our formal interpretation of the PFC system satisfies the availability requirements.

Initial Configuration	Final Configuration: 333	332	322	222	330	320	220	300	200	000
333		P:$1.80 * 10^{-06}$ M:0,27,21,73	P:$1.08 * 10^{-12}$ M:0,0,252,279	P:$2.16 * 10^{-19}$ M:0,0,0,783	P:$4.02 * 10^{-07}$ M:0,28,82,4	P:$4.83 * 10^{-13}$ M:0,0,560,2320	P:$1.45 * 10^{-19}$ M:0,0,0,2772	P:$5.06 * 10^{-14}$ M:0,0,208,1440	P:$4.55 * 10^{-20}$ M:0,0,0,3147	P:$1.21 * 10^{-20}$ M:0,0,3,576
332	P:0.00 FALSE		P:$1.80 * 10^{-06}$ M:0,30,0,100	P:$1.08 * 10^{-12}$ M:0,0,270,162	P:$2.20 * 10^{-06}$ M:0,55,1,28	P:$4.03 * 10^{-07}$ M:0,34,379,640	P:$4.83 * 10^{-13}$ M:0,0,648,3581	P:$5.33 * 10^{-13}$ M:0,0,658,2264	P:$5.06 * 10^{-14}$ M:0,0,255,6902	P:$1.57 * 10^{-19}$ M:0,0,3,5472
322	P:0.00 FALSE	P:0.00 FALSE		P:$1.80 * 10^{-06}$ M:0,30,0,104	P:0.00 FALSE	P:$2.20 * 10^{-06}$ M:0,61,1,37	P:$4.03 * 10^{-07}$ M:0,34,406,565	P:$1.61 * 10^{-12}$ M:0,0,916,510	P:$5.34 * 10^{-13}$ M:0,0,768,4386	P:$7.26 * 10^{-19}$ M:0,0,150,6675
222	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE		P:0.00 FALSE	P:0.00 FALSE	P:$2.20 * 10^{-06}$ M:0,61,1,37	P:0.00 FALSE	P:$1.61 * 10^{-12}$ M:0,0,1023,566	P:$1.46 * 10^{-18}$ M:0,3,0,8400
330	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE		P:$1.80 * 10^{-06}$ M:0,30,51	P:$1.08 * 10^{-12}$ M:0,0,270,657	P:$4.02 * 10^{-07}$ M:0,28,88,412	P:$7.24 * 10^{-13}$ M:0,0,684,2475	P:$1.61 * 10^{-13}$ M:0,0,211,4192
320	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE		P:$1.80 * 10^{-06}$ M:0,30,60	P:$2.20 * 10^{-06}$ M:0,55,37,31	P:$4.03 * 10^{-07}$ M:0,34,583,280	P:$1.42 * 10^{-12}$ M:0,0,958,3499
220	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE		P:0.00 FALSE	P:$2.20 * 10^{-06}$ M:0,61,40,70	P:$3.86 * 10^{-12}$ M:0,3,1240,40
300	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE		P:$2.70 * 10^{-06}$ M:0,39,0,36	P:$7.02 * 10^{-07}$ M:0,25,220,1
200	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE		P:$3.40 * 10^{-06}$ M:0,64,1,1
000	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	

Table 30.5: Boeing 777 FBW modes reachability analysis

The result of the reachability matrix analysis, reported in Table 30.5, demonstrates that the majority of the fault trees are FALSE, representing that the final configuration cannot be reached by that specific initial mode. This result confirms the monotonic degradation of the PFC lanes e.g., once the configuration 222 is reached then only 222, 220, 200, or 000 will be reachable. This result extends what is represented in Figure 30.8 by adding probabilities to the transitions that link each pair of states.

Table 30.5 may erroneously recall a Markov Chain transition matrix, however the semantics of the two are not directly comparable. In fact, in case of a Markov Chain the value of the cell [333,332] would express the probability of going from 333 to 332 in 1 step. Moreover, a Markov Chain transition matrix requires that the sum of each row to be equal to 1, which does not apply for Table 30.5 due to the fact that we are labeling with probabilities only the fault events.

30.4 Airbus A330 Flight Computers

This section shows an application of the techniques for the verification of redundant system architectures introduced in Part IV. The case-study that we take into account is the Primary Flight Computer (PFC) architecture of the Airbus A330, described in [114, 10].

30.4.1 System Description

The architecture of the Fly-by-Wire system of the Airbus A330 follows an extension of the Dual Modular Redundancy, which acts in an "hot-standby" fashion. This type of architectural redundancy is characterized by two sub-systems: primary and secondary. More specifically, the latter takes over on the computation when the primary is disabled, due to a detected system malfunctioning. In the PFC of the A330, both primary and secondary systems have the capability of interpreting the pilot's objectives, and translate them into actuators command. Airbus A330 primary and secondary systems are respectively composed of three and two computers. The architecture under analysis, shown in Figure 30.9, integrates the computer systems with additional components managing the interaction between them. In fact, only one computer at a time, called *master*, will be active and computing the actuators command, while all the others are supervising it by providing a feedback signal. The interpretation that we applied centralizes this functionality into the *Hub Feedback* component, which routes the feedback to the current active computer. Internally, both primary and secondary computers have a switch component that connects the command to the output of the current active computer. The *Flight Control Data Concentrators* (FCDC) component operates as a switch at the sub-systems level, with the difference that it is implemented with a dual redundancy approach in order to avoid single point of failures.

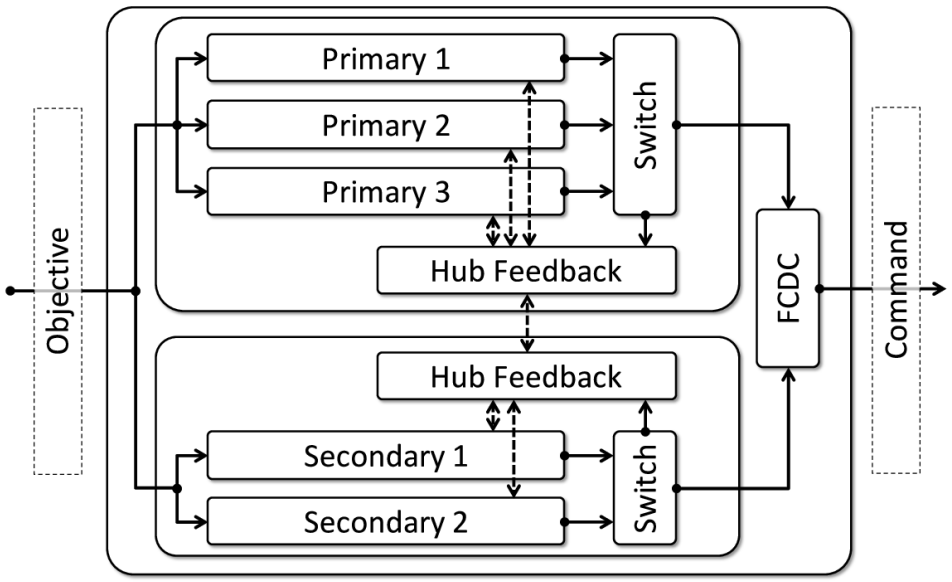


Figure 30.9: Airbus A330 FBW Architecture

Power System

Power system and PFC have a tight dependency, thus even the former requires a design that assures fault tolerance and reliability.

The main source of power in a modern aircraft is represented by the engines, and the Permanent Magnet Generators (PMG) are the components that transform kinetic energy into electrical power. Some aircraft integrate the engines PMGs with Auxiliary Power Units (APU). Those components, based on an internal combustion engine, do not produce thrust but are directed only to generate electrical power. In the case of an Airbus A330, the system relies on a single PMG for each engine, and an APU installed on the tail of the aircraft. In this setting, a single PMG or APU is sufficient to supply the entire aircraft.

As described earlier in this Chapter, modern aircraft have no mechanical backup and the loss of electrical power would result in a catastrophic condition. In order to mitigate those event, the power system is integrated with multiple batteries that provide enough power when no generators

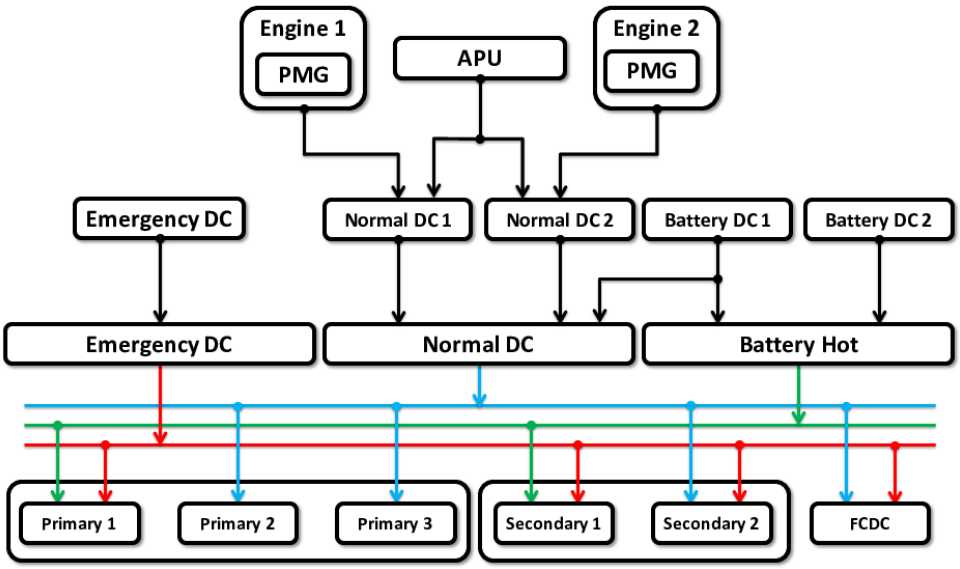


Figure 30.10: Airbus A330 Power System

are available. Figure 30.10 shows the power system architecture implemented in the Airbus A330, which is composed of 2 PMGs, 2 batteries, an APU, and an additional emergency generator linked to the hydraulic system. The power system supplies the PFC via 3 disjoint lanes: *Normal DC*, *Battery Hot*, and *Emergency DC*. The power distribution, as depicted in Figure 30.10, is designed in order to provide at most two sources for each main computer and FCDC system.

Computers Implementation

The implementation of the Airbus A330 computers, other than the five replicas, adds an additional level of redundancy. In fact, each computer integrates two CPUs that elaborate the pilot's objectives. Those components differ from the fact that the first one computes the actuators commands, and the second one monitors its activity by comparing the output value with the result of a local computation. The monitoring activity is fundamental in this architecture, because one single primary/secondary

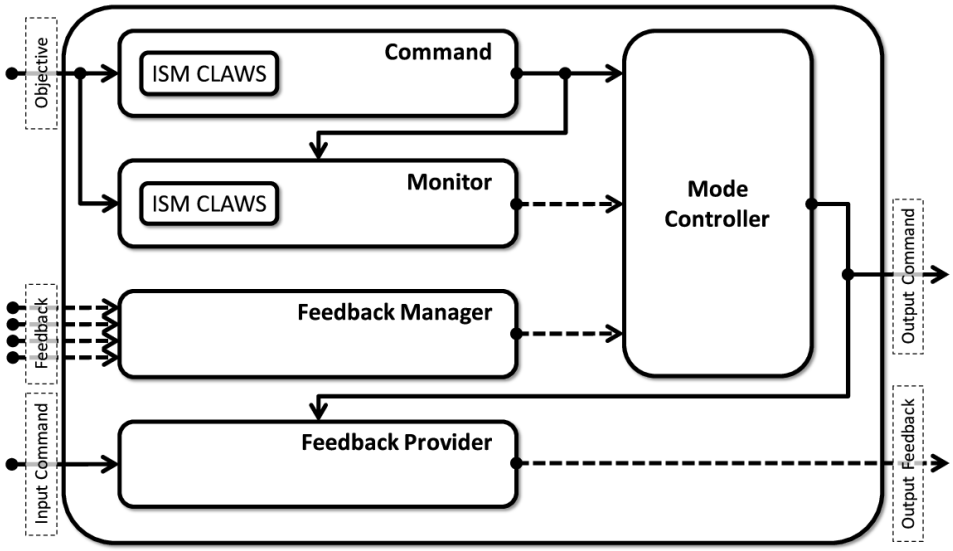


Figure 30.11: Airbus A330 Primary/Secondary Computer

computer will be active at the same time. Due to this fact, the Airbus imposed there be four different cpu and software designs, supervised and developed by four disjoint engineering teams. The implementation diversity has been categorized into i) primary command, ii) primary monitor, iii) secondary command, and iv) secondary monitor.

As described earlier in this section, each computer has to provide a feedback to the current master computer, as well as manage the feedback coming form the other ones i.e., in case of acting as a master. Figure 30.11 shows that these functionalities are in charge of two separate components called respectively *Feedback Provider* and *Feedback Manager*. The latter's behavior is similar to a monitor component, in fact it compares the master's command with the local command, and the output feedback represents the value of the equality between the input signals. The feedback manager collects all input feedbacks into a single one, which is then provided to the *Mode Controller*. This component is the one devoted to deciding whether the command should transferred to the output or not, according with the pass/fail signals coming from monitor and feedback manager components.

Mode	Primary Computers	Secondary Computers	Configuration
Normal	3/3	2/2	P3 S2
Normal-S1	3/3	1/2	P3 S1
Normal-S0	3/3	0/2	P3 S0
Normal-D	2/3	2/2	P2 S2
Normal-D-S1	2/3	1/2	P2 S1
Normal-D-S0	2/3	0/2	P2 S0
Alternate	1/3	2/2	P1 S2
Alternate-S1	1/3	2/1	P1 S1
Alternate-S0	1/3	2/0	P1 S0
Direct	0/3	2/2	P0 S2
Direct-D	0/3	1/2	P0 S1
Dead	0/3	0/2	P0 S0

Table 30.6: Airbus A330 FBW modes

System Reconfiguration

The PFC of the Airbus A330, as every redundant systems, has multiple operational modes that are selected according to the level of system degradation. Those levels are imposed by the failure occurrence, and follow a predefined pattern when multiple choices are possible. Under normal conditions i.e., when no failures are detected, the PFC selects a master out of the three primary computers. The role of the master is to elaborate the pilot's objectives into actuators command. This turns out to have one computer that performs the actual PFC activity, and 5 (one local to the master computer, and 4 from the other primary/secondary computers) that act as monitors. When a single feedback signal becomes FALSE, meaning that a mismatch has been detected, the internal mode controller disables the entire computer. This results into the promotion to master of one the primary computers, if no primary one is available, then a secondary will be selected. The remaining computers will act as feedback monitors. This re-

Component	# Failures	
	Visible	**Hidden**
Power System	6	0
Primary Computers	33	18
Secondary Computers	20	10
Common Cause Implementation	0	4
FCDC	2	2
Total	61	34

Table 30.7: Components Failures

configuration approach delivers a list of possible operational modes, which are described in Table 30.6.

30.4.2 System Faults Definition

As listed in Table 30.7, the resulting formal model of the Airbus A330 PFC has a total of 95 possible faults, where each of them is assigned to a probability of occurrence. In case of a visible failure, the probability is set to 10^{-7} for one hour of operation, while for hidden failures the value is set to 10^{-10}. Those probabilities represent an "extremely improbable" and an "extremely remote" event.

As for the design of the Boeing 777 PFC, the Airbus A330 differentiates the design of the computational modules. In this case, command and monitor computations are based on different designs, and the diversity is applied also between primary and secondary computers. This approach entails a total of four different implementation designs. In practice, Commands and Monitors are softwares that compute the actuators commands according with pilot's objectives, thus their probability are set to 0.0 because they are not quantifiable. Even though the software implementation events are not taking part in the probability computation, they will be still

considered and analyzed in the minimal cutsets computation.

30.4.3 Formal Modeling and Model Validation

The architecture description of the Airbus A330 PFC is modeled by relying on the techniques described in Chapter 20. In this case, we described the system as a combinatorial model. More specifically, in absence of faults the outputs of the PFC are a function of the current inputs, and not dependent on the previous inputs or computed values. In fact, under this assumption the whole PFC is equivalent to a single "ISM CLAWS" module. The extension with failures adds a non deterministic behavior to the system, however the PFC's outputs remain dependent only on current inputs and state variables.

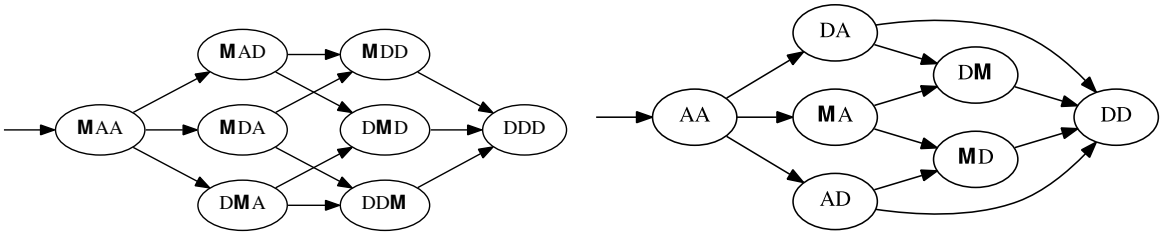
The Airbus A330 PFC system modeling required a total amount of 20 man-hours[3]. The resulting OCRA model is composed of 586 lines of code, that turned out to generate a miter construction in SMV language of 848 lines. The state machine generated has 341 state variables (113 real and 228 Boolean) where 95 of those are fault variables, and 10 uninterpreted functions.

This analysis resulted in the definition of a set of 174 invariant properties.

30.4.4 Abstract Model Analysis

The Abstract Model Analysis of this system is performed on the model with additional constraints that bound to one the number of component failures for each enrolling of the transition relation. This extension allows us to have an explicit representation of the transitions between each reachable state.

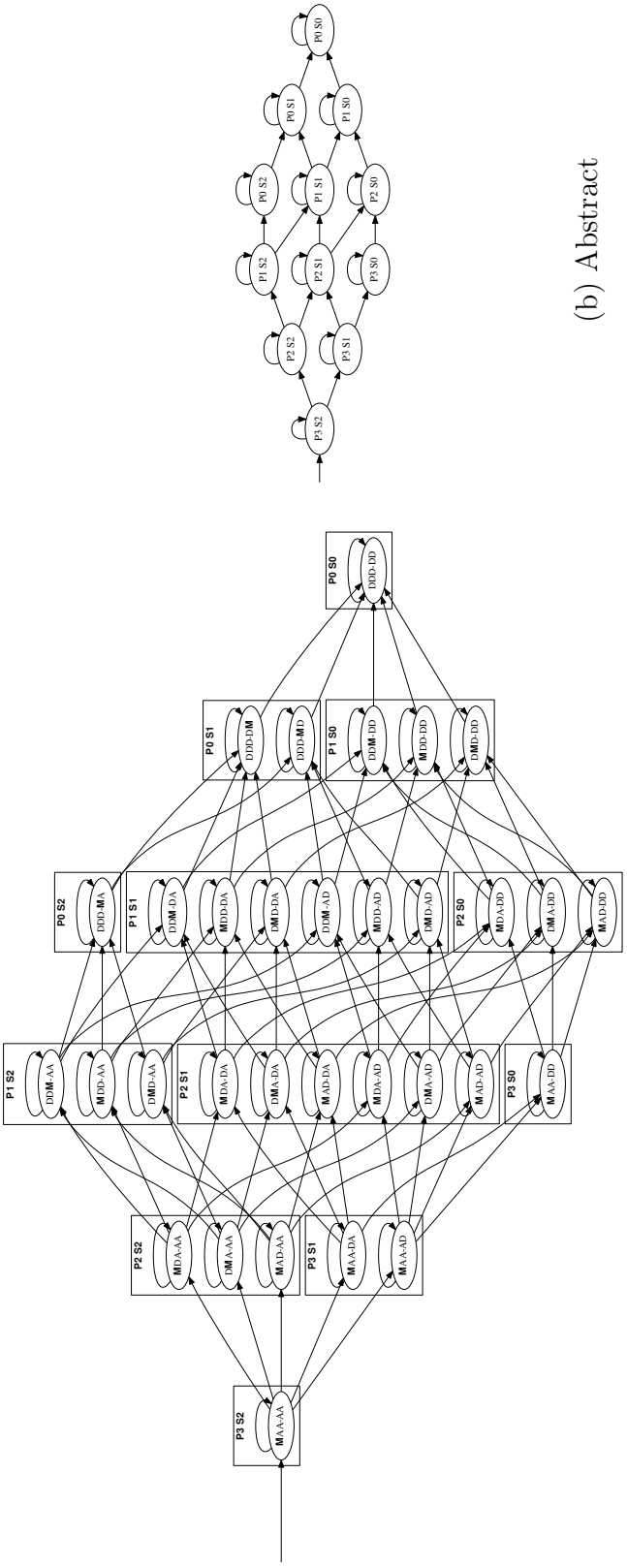
[3]This does not consider the time needed to finalize the system design.



(a) Primary Computers (b) Secondary Computers

Figure 30.12: Airbus A330 FBW reachable modes

The set of predicates chosen for the abstract model analysis extract the operational modes of primary and secondary computers. Specifically, each computer can operate in three different modes: Active/Available (A), Disabled (D), or Master (M), but just one out of five computers can be in master mode. Figure 30.12a shows how the primary computers operational modes evolve. As expected, a master is selected when at least one computer is active . This represents the actual behavior of the system, in fact a secondary master should be chosen only when no primaries are available. The result for the primary computers analysis is aligned with the secondary ones, as shown in Figure 30.12b. In fact, the secondary computers allow for not being selected as master when at least a primary one is available.



(a) Full

(b) Abstract

Figure 30.13: Airbus A330 FBW reachable modes (w.r.t., Table 30.6)

The synchronous product of the two state machines in Figure 30.12 represents the operational modes of the entire system. The resulting abstract state machine can be obtained from the analysis that expresses both predicates i.e., for primary and secondary computers. The outcome of this analysis is shown in Figure 30.13, and each state is associated to the corresponding mode as in Table 30.7.

30.4.5 Fault Tree Analysis

The reliability evaluation of the PFC A330 is carried out by applying the techniques described in Chapter 20. This allowed us to evaluate under which condition the system provides a wrong result. This analysis is performed on all possible initial configurations of the system i.e., from P3 S2 to P0 S0. Table 30.8 lists the results for this analysis, by showing the number of minimal cutsets for the cardinalities 0, 1, 2, and 3, with the computed probability. The bold values express the number of fault configurations containing common cause implementation failures. We remark that if the analysis returns TRUE it means that the property does not hold even in the nominal case i.e., without triggering any faults. Clearly, this is the case when all computers are disabled (configuration P0 S0).

The result of this analysis shows a high level of system reliability for the first 6 configurations in Table 30.8, in fact the resulting probability of obtaining a wrong output is lower than 10^{-11}. Notably, the (common cause) implementation failures have lower impact on those configurations, due to the fact that at least one of primary or secondary computer is available.

The result of the reachability matrix analysis, reported in Table 30.9, demonstrates that the majority of the fault trees are FALSE, representing that the final configuration cannot be reached by that specific initial mode. This result confirms the monotonic degradation of the PFC channels e.g.,

Initial Configuration	Minimal Cutsets 0	1	2	3	Probability
P3 S2	0	0	**7** 3	**356** 26	$2.00 * 10^{-14}$
P2 S2	0	0	**7** 4	**1133** 51	$2.00 * 10^{-14}$
P3 S1	0	0	**55** 3	**146** 130	$2.00 * 10^{-14}$
P1 S2	0	0	**75** 5	**348** 4871	$2.00 * 10^{-14}$
P2 S1	0	0	**55** 4	**923** 9499	$2.00 * 10^{-14}$
P1 S1	0	0	**123** 821	**138** 287	$2.68 * 10^{-12}$
P3 S0	0	**2** 0	7	1512	$2.00 * 10^{-14}$
P0 S2	0	**2** 0	144	33	$5.11 * 10^{-13}$
P2 S0	0	**2** 0	397	22	$1.22 * 10^{-12}$
P0 S1	0	**2** 24	7	1	$1.40 * 10^{-6}$
P1 S0	0	**2** 34	3	1	$1.90 * 10^{-6}$
P0 S0	TRUE				$1.00 * 10^{+00}$

Table 30.8: Airbus A330 FBW: wrong output value

once the configuration P1 S1 is reached then only P0 S1, P1 S0, or P0 S0 will be reachable. This result shows a different perspective of the abstract model analysis in Figure 30.13, where each cell with a positive probability represents a transition in Figure 30.13b.

Initial Configuration		Final Configuration															
		P3 S2	P2 S2	P3 S1	P1 S2	P2 S1	P1 S1	P3 S0	P0 S2	P2 S0	P0 S1	P1 S0	P0 S0
P3 S2			P:1.60 * 10^-06 M:0,33,1	P:1.20 * 10^-06 M:0,22,1,8	P:8.52 * 10^-13 M:0,2,320,20	P:1.92 * 10^-12 M:0,0,726,57	P:1.02 * 10^-18 M:0,0,44,7054	P:3.60 * 10^-13 M:0,2,100,10	P:1.50 * 10^-19 M:0,2,0,100	P:5.77 * 10^-19 M:0,0,66,3302	P:0.00 M:0,0,40,26	P:0.00 M:0,0,4,840	P:0.00 M:0,0,4,236
P2 S2	P:0.00 FALSE			P:0.00 FALSE	P:1.60 * 10^-06 M:0,33,1	P:1.20 * 10^-06 M:0,22,1,21	P:1.92 * 10^-12 M:0,0,726,193	P:0.00 FALSE	P:8.52 * 10^-13 M:0,2,320,20	P:3.60 * 10^-13 M:0,2,100,10	P:6.82 * 10^-19 M:0,0,40,6426	P:5.77 * 10^-19 M:0,0,66,3302	P:0.00 M:0,0,4,876
P3 S1	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE		P:0.00 FALSE	P:1.60 * 10^-06 M:0,33,1,1	P:8.52 * 10^-13 M:0,2,320,20	P:1.20 * 10^-06 M:0,22,1,8	P:0.00 FALSE	P:1.92 * 10^-12 M:0,0,726,57	P:1.50 * 10^-19 M:0,2,0,1100	P:1.02 * 10^-18 M:0,0,44,7054	P:0.00 M:0,0,48,2
P1 S2	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE		P:0.00 FALSE	P:1.20 * 10^-06 M:0,22,7,21	P:0.00 FALSE	P:1.60 * 10^-06 M:0,33,1	P:0.00 FALSE	P:1.28 * 10^-12 M:0,0,660,451	P:3.60 * 10^-13 M:0,2,100,64	P:5.77 * 10^-19 M:0,0,66,3896
P2 S1	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE		P:1.60 * 10^-06 M:0,33,1,1	P:0.00 FALSE	P:0.00 FALSE	P:1.20 * 10^-06 M:0,22,1,21	P:8.52 * 10^-13 M:0,2,320,20	P:1.92 * 10^-12 M:0,0,726,193	P:1.02 * 10^-18 M:0,0,48,7682
P1 S1	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE		P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:1.60 * 10^-06 M:0,33,1,1	P:1.20 * 10^-06 M:0,22,7,21	P:1.92 * 10^-12 M:0,0,792,59
P3 S0	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE			P:0.00 FALSE	P:1.60 * 10^-06 M:0,33,1,1	P:0.00 FALSE	P:8.52 * 10^-13 M:0,2,320,20	P:1.50 * 10^-19 M:0,2,0,1100
P0 S2	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE		P:0.00 FALSE	P:8.01 * 10^-07 M:0,20,13,1	P:0.00 FALSE	P:3.61 * 10^-13 M:0,2,118,10
P2 S0	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE		P:0.00 FALSE	P:1.60 * 10^-06 M:0,33,1,1	P:8.52 * 10^-13 M:0,2,320,20
P0 S1	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE		P:0.00 FALSE	P:1.20 * 10^-06 M:0,24,1,1
P1 S0	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE		P:1.60 * 10^-06 M:0,33,1,1
P0 S0	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	P:0.00 FALSE	

Table 30.9: Airbus A330 FBW modes reachability analysis

30.5 Related Works

In [122, 96, 81, 110, 10, 114] the redundant Fly-by-Wire architectures of Boeing 777 and Airbus A330 are described. However, none of them apply formal analysis to the FBW systems.

[17] describes the technique applied to the development of the primary flight computer of the Airbus A380. In particular, this paper shows how the design space has been analyzed in order to select the best architecture out of the 10^{59} possible ones. This work describes a branch and bound approach that relies on a user defined architecture-to-cost mapping in order to prune the search space.

The approach described in [88] solves the problem of components displacement and assembly, in addition to an optimization check that excludes the solutions that are not optimal.

30.6 Conclusion and Future Works

Modern aircraft reached a level of complexity and weight that pilots cannot maneuver them just via pure mechanical systems. Therefore, current aircraft rely on a Fly-by-Wire system that listens to pilot's requests, translates them into electrical signals, computes the actuators controls, and it finally converts them into hydraulic force to move the flight surfaces. Nowadays, the Fly-by-Wire system is the main system that drives the aircraft flight surfaces, and very few modern aircraft rely on a mechanical backup. Given these aspects, guaranteeing safety and reliability of Fly-by-Wire systems is fundamental in aircraft design.

As with many safety critical systems, FBW systems rely on redundancy in order to guarantee high levels of safety and reliability. The analyses introduced in Part IV are tailored to aid the development of redundant

architectures, thus they can be applied to support the development of FBW system designs.

In this Chapter we have provided an extensive formal analysis of the redundant flight computer of Boeing 777 and Airbus A330. The resulting analysis have shown the practical capabilities of the techniques described in Part IV, with the application of the SMT-based minimal cutsets computation engine proposed in Chapter 8.

Future works will enrich the modeling with contracts [58], in order to enable for the compositional verification, and compare with the contract-based safety assessment techniques in Part III. Moreover, further analysis will be directed to synthesize the best architecture according to a given objective function.

31

Formal Design and Safety Analysis of AIR6110 Wheel Brake System

This Chapter reports an application of the contract-based safety analysis approach on the Wheel Brake System case study described in the SAE Aerospace Information Report 6110 [109] "Contiguous Aircraft/System Development Process Example". This work is described in [36], and it applies the techniques in Part III to a large scale system description by comparing the result of the compositional approach to the monolithic one. In this Chapter we provide an extract of the most relevant parts for this Thesis.

31.1 The Airspace Information Report 6110

The Society of Automotive Engineers (SAE) International is a United States-based professional association for engineering professionals. Most notably, SAE International contributes on the development of standards for guiding design and development in automotive, aerospace, and commercial vehicles fields.

The Aerospace Recommended Practice (ARP) 4754A [108] and 4761 [107] are documents developed by SAE that define process and methodologies for assuring safety and reliability in avionic domain. The

practices prescribed by these documents are recognized by the Federal Aviation Administration (FAA) as acceptable means for showing compliance with federal regulations [7, 8], and have been used by the industry of the field for years. In 2011, SAE released the Aerospace Information Report 6110 (AIR6110) document that, following the principles defined in ARP4754A and ARP4761, describes the development of several subsystems of a hypothetical aircraft. The AIR6110 focuses on the Wheel Brake System (WBS) of a passenger aircraft, capable of transporting between 300 and 350 passengers, and with an average flight duration of 5 hours. The WBS under analysis is a hydraulic brake system that provides the primary stopping force during landing operations.

31.2 Overview of the WBS

31.2.1 WBS architecture and behavior

An overview of the Wheel Braking System architecture is shown in Figure 31.1. The WBS drives the braking force to 4 landing gears, two on the left side and two on the right, and each landing gear is composed of two wheels. The wheel braking system is composed of three sub systems: hydraulic, electric, and mechanic. The hydraulic system provides the source of power that can be converted into braking force, and the electrical one drives the braking operations. The mechanical system operates as backup in case of electrical malfunctioning.

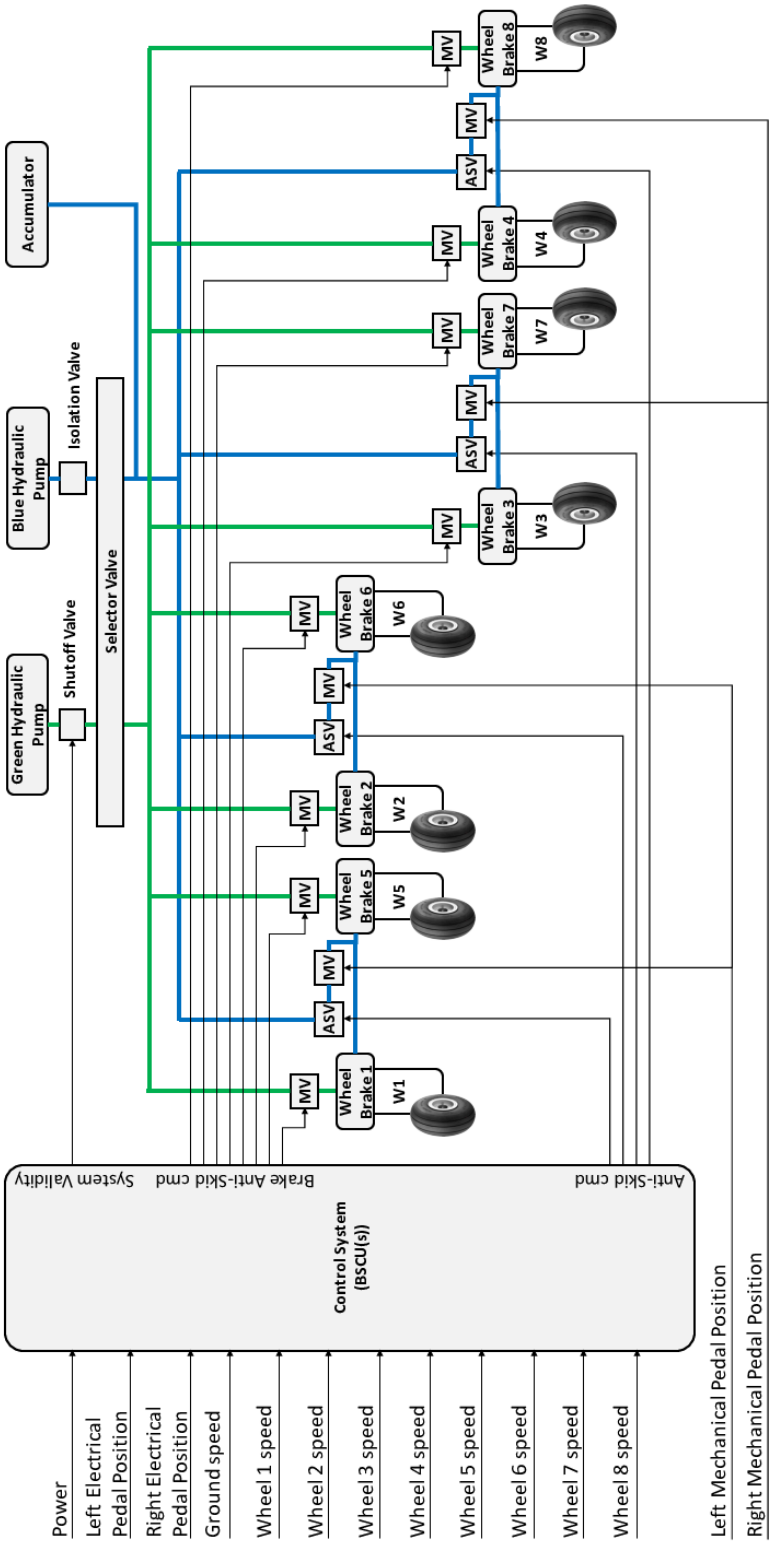


Figure 31.1: WBS 6110 Architecture

The wheel braking system is a redundant system, and it can operate in three different operational modes, according with the current sub-systems availability. In absence of failures the system acts in *normal* mode, thus the hydraulic power is provided independently to each wheel through the green hydraulic system. A separate meter valve for each wheel controls the hydraulic flow, and it is controlled by the Braking System Control Unit (BSCU). The BSCU is able to apply independent brake force to each wheel, and therefore to provide anti-skid protection by analyzing ground speed, wheel speed, and brake commands. In case of malfunction of the green hydraulic, the system switches to the *alternate* mode. In this operational mode the hydraulic force is provided to each landing gear by the blue system, and the switch from green to blue is operated by a selector valve. The alternate mode does not allow for an independent control of each wheel, but in unison with the wheel of the same landing gear. The blue hydraulic system is composed of four anti-skid shutoff valves and four meter valves. The anti-skid valves are controlled by the BSCU, which supervises the flow to the meter valves in order to avoid wheel skid. The meter valves operates directly on the wheels brake, and they are controlled by the pilot's left and right pedals. In case of failure of the blue pump, the WBS guarantees an additional level of reliability. In fact, under these circumstances an accumulator takes over by providing sufficient pressure to brake the aircraft. Moreover, an isolation valve placed before the pump prevents pressure from flowing back to the blue pump.

Sensors for the pedal position and the wheels' angular speed are also part of the system, though not represented in the diagram in Figure 31.1 for the sake of clarity.

31.2.2 System Requirements

The AIR6110 document contains a set of safety requirements on the expected probability of an unwanted event occurrence e.g., "the loss of all wheel braking shall be extremely remote". The case study in [36] focuses on the verification of five safety requirements:

S18-WBS-R-0321 Loss of all wheel braking (unannunciated or annunciated) during landing or rejected take off shall be extremely remote

S18-WBS-R-0322 Asymmetrical loss of wheel braking coupled with loss of rudder or nose wheel steering during landing or rejected take off shall be extremely remote

S18-WBS-R-0323 Inadvertent wheel braking with all wheels locked during takeoff roll before V1 shall be extremely remote

S18-WBS-R-0324 Inadvertent wheel braking of all wheels during takeoff roll after V1 shall be extremely improbable

S18-WBS-R-0325 Undetected inadvertent wheel braking on one wheel w/o locking during takeoff shall be extremely improbable Intuitively, a safety requirement associates the description of an undesirable behaviour or condition (e.g. "inadvertent wheel braking") with a lower bound on its likelihood, according to terminology (e.g. "extremely improbable") defined in [9].

31.3 Formal Modeling

The system described in the [36] has been modeled considering to follow a development process with architectural refinement. Therefore, the

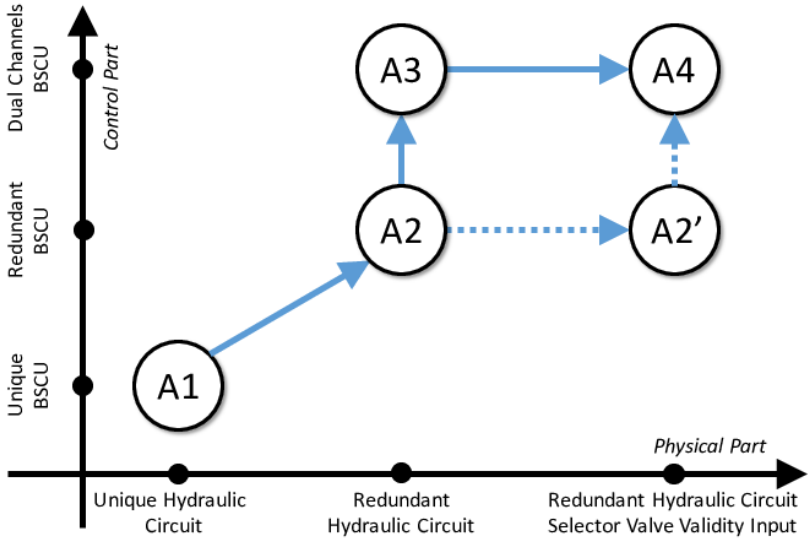


Figure 31.2: Formal Models

modeling considers five different revisions of the original architecture as in Figure 31.2. The *Arch1* considers only green hydraulic and the control unit, thus it does not satisfy the safety requirements. This motivated the definition of *Arch2*, which extended with the additional blue hydraulic circuit and a dual redundancy in the BSCU component. Further analysis to identify less expensive and easier to maintain alternative to *Arch2*, but answering to the same safety requirements, opened up for the definition of *Arch3*. In fact, this architecture moves from two BSCUs to a single dual-channeled BSCU. Moreover, *Arch4* extends *Arch3* in order to met a safety requirement addressing mutual exclusion of the operating modes of the WBS. Only the physical system is modified, by adding an input to the selector valve corresponding to the validity of the control system and moving the accumulator in front of the selector valve.

An additional architecture called *Arch2bis* (A' in Figure 31.2) has been modeled, and it is based on the control system architecture of *Arch2* and the physical system architecture of *Arch4*. The purpose is to show that it is possible to detect the issue that motivated the change to *Arch4* earlier

		Arch1	*Arch2*	*Arch2bis*	*Arch3*	*Arch4*
Architecture Decomposition	Total components types	22	29	29	30	30
	Leaf components types	15	20	20	20	20
	Total components instances	100	168	168	169	169
	Leaf components instances	79	143	143	143	143
	Max depth	5	5	5	6	6
	Contracts	121	129	129	142	142
System Implementation	Properties	199	291	291	304	304
	Bool. Vars	31	79	79	79	79
	Enum Vars	55	88	88	88	88
Extended System Implementation	Failure modes	28	33	33	33	33
	Fault variables	170	261	261	261	261
	Bool. Vars	74	156	156	156	156
	Enum Vars	184	311	311	311	311

Table 31.1: Models Statistics

in the design process at *Arch2*.

All five architectures are described using the OCRA contract-based design, and follows the process described in Part V (without the redundant architecture analysis). More specifically, each architecture is hierarchically decomposed in sub-components defined with input and output ports. Each component has a set of contracts that are refined by its sub-components contracts, while the leaf components are refined with a state machine implementation. This system decomposition, called nominal, is then verified by relying on OCRA *check refinement* and *check implementation*.

The nominal model is then extended to represent the faulty behavior. This phase is automatically performed on the contract decomposition, with the contract-based safety assessment technique, and on the leave implementations with fault injection. The safety analysis is then performed with the integration of OCRA and xSAP, able to provide a hierarchical fault tree for each top level system contract.

Property	**MCS cardinality**					
	1	2	3	4	5	> 5
S18-WBS-R-0321	0	6	627	629	T.O.	T.O.
S18-WBS-R-0322-left	2	2	203	46287	T.O.	T.O.
S18-WBS-R-0322-right	2	2	203	46287	T.O.	T.O.
S18-WBS-R-0323	0	0	0	0	0	T.O.
S18-WBS-R-0324	0	1	0	2	8729	T.O.
S18-WBS-R-0325-wheel1..8	9	12	2596	0	0	0
cmd implies braking w1	13	30	7428	3815	1768	0
braking implies cmd w1	10	24	2647	4530	59	0

Table 31.2: Fault Tree Analysis results on *Arch4* monolithical model

The compositional approach on a contract-based system can be also integrated with a monolithical analysis. More specifically, it is possible to combine all leaf implementing a single state machine. OCRA provides this functionality, and it generates a single SMV file representing the whole architecture. Standard validation and verification can be then applied on the monolithical model, as well as fault tree analysis via minimal cutsets computation. In [36] a comparison between monolithical and compositional approaches has been analyzed. Statistical information on the resulting models is reported in Table 31.1.

31.4 Safety Analysis

When the model under analysis reaches a significant size (e.g., ∼600 state variables with ∼300 fault variables) the analysis of the minimal cutsets becomes impractical. This limitation seems to has more impact on the analysis of the results than on the computational power needed to extract that information. Table 31.2 reports the results of the minimal cutsets computation on *Arch4*, by applying the IC3-based techniques described

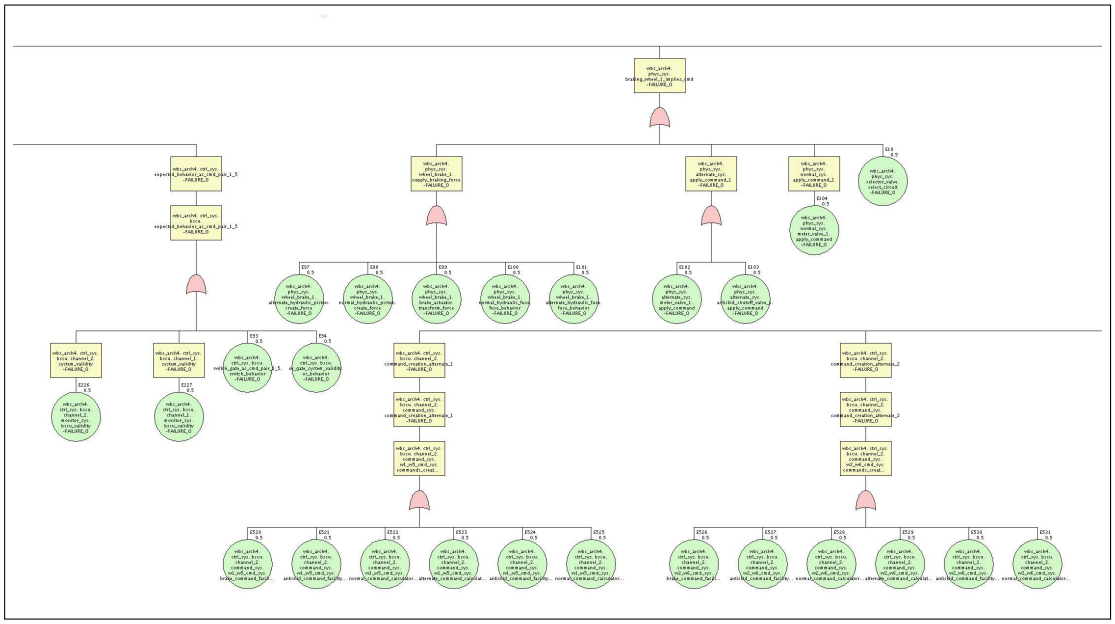
in Part II. Considering the case of S18-WBS-R-0322 property, a manual inspection of more than 46000 MCS is completely out of reach. However, assuming that the lower the cardinality of a MCS is, the higher it is its significance, this analysis become relevant when looking for single or double point of failures in the system.

The monolithical fault tree analysis performs the evaluation without any information on the system architecture under analysis. This results in the impossibility to break the symmetry that characterizes almost all redundant systems. For instance, we can consider a simple example when a top level event can be reached when either of C1 and C2 components fail, and where each C1, C2 is composed of two fundamental parts. In this case, the monolithical analysis would provide the cutsets {{C1.pt1, C2.pt1}, {C1.pt1, C2.pt2}, {C1.pt2, C2.pt1}, {C1.pt2, C2.pt2}}. Extending this example with 8 components, as in the WBS case study, the resulting MCS would be exponentially bigger i.e., 256.

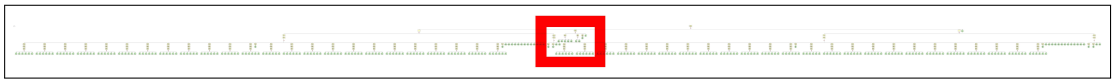
The contract-based safety analysis, differently from the monolithic approach, performs the fault tree analysis by inspecting the hierarchical decomposition of the system. Thus, this approach is not affected by the combinatorial explosion in the minimal cutsets extraction. Moreover, the contract-based approach leverages on a compositional verification and avoids unnecessary analysis of the duplicated parts.

Taking into account the results summarized in Table 31.2, the total number of MCS for the property S18-WBS-R-0325-wheel1 are ∼2600, which turns out to generate a tree with ∼7800 leaves (i.e., $(9 * 1) + (12 * 2) + (2596 * 3)$). Figure 31.3 reports the fault tree obtained with contract-based safety assessment on the same property. Figure 31.3b represents the full fault tree, while Figure 31.3a shows a zoomed part represented by the red square in 31.3b. The representation of the full fault tree hierarchically organized is still quite big, however its size of ∼400 leaves is significantly

lower than its monolithical counterpart. We remark that the fault tree in Figure 31.3 is automatically generated out of the OCRA system decomposition, and represented via OpenFTA tool [98]. The advantages of the compositional approach over the monolithical one are not only relevant to the quality of the produced artifact, but also on the computational time. In fact, the complete monolithical fault tree analysis on *Arch4* takes ∼1000 minutes to complete (without considering the time outs), while the compositional one requires only 1 minutes and 30 seconds. However, and as discussed in Chapter III, the minimal cutsets computed on the hierarchical decomposition can be an over-approximation of the one resulting from the monolithic analysis. This outcome is directly dependent on the tightening of the components (contract) refinement, and it is part of the standard practice in safety analysis.



(a) Extract of the Fault Tree (Red Square)



(b) Full Representation

Figure 31.3: Example of Resulting Fault Tree

31.5 Conclusion

This Chapter provides an overview of the results achieved in [36], by applying the techniques of contract-based safety assessment to an aircraft wheel braking system.

The analysis of the WBS, which is a complex real-world case-study, has demonstrated the significant improvement reached by the IC3 based minimal cutsets computation introduced in Part II. This technique, integrated with the contract-based safety analysis (Part III), allowed for an automated and efficient generation of hierarchically organized fault trees.

32

Thesis Conclusion

In this Thesis we propose a novel approach to the model-based safety assessment. The techniques that we have introduced are able to overcome the problems of scalability, structured fault tree generation, and integration with reliability analysis that characterized previous approaches. The application of our techniques to a set of real-world case studies demonstrated its capability of supporting the analysis of safety critical systems.

The integration of standard safety analysis with contract-based design enables for automatically generated multi-layered artifacts such as Fault Trees. Moreover, it overcomes the previous limitations by providing a seamless integration with the analysis of the nominal model, and natively support model refinement. In fact, the application of this technique to the Avionic Recommended Practice [108, 107], and Aerospace Information Report [109] demonstrated its ability to be compliant with the current prescribed standards.

The reliability analysis, specialized for redundant architectures, extends the process outlined by the contract-based safety assessment. In fact, it integrates the standard architecture refinement with a set of techniques tailored to improve the reliability aspects of a redundant system. Furthermore, its extension with predicate abstraction provides a performance

improvement of two orders of magnitude, enabling the possibility to analyze architectures with hundreds of redundant components.

At the basis, the application of modern IC3-based symbolic verification allowed us to reach significant improvements in the minimal cutsets computation routines. The experimental evaluation comparing this technology with the previous ones has shown a remarkable increase of case-studies instances solved in the time limit. The minimal cutsets computation is the formal analysis engine of all others model-based safety analysis, and this improvement provided a performance gain in all applications.

The main target of this Thesis was to provide practical improvements in the design process of safety critical systems. This goal can be only achieved with a concrete tool support, whose implementation is guided by the application to real world case studies. We have reached this target by defining a symbolic model checking based platform, implemented into a set of specialized tools such as nuXmv [97], xSAP [27], and OCRA [54]. The results obtained in projects like the formal analysis of the next generation of the air traffic control, and the aircraft directed wheel braking system have demonstrated the significant capabilities of the proposed methodology.

For the future, we will extend the current approach in different directions. For the contract-based safety assessment, we will support the generation of FMEA tables, by reorganizing the results obtained by the monotonic Fault Tree Analysis. Moreover, the fault injection at contract level will be refined by expressing more detailed faulty behaviors. Regarding the redundant architecture analysis, we intend to add an automated synthesis of the most reliable architectures that do not exceed cost and weight constraints. More in general, we will investigate techniques able to optimize a given function, while preserving the original behavior of the system. The new algorithms for the minimal cutsets computation are currently able to deal with very large problems, but this holds only when failure variables

satisfy the monotonicity assumption. In fact, at the current stage there is no efficient ways of performing parameter synthesis with such a huge number of components failure. For the future, we will investigate alternative approaches able to guarantee a level of performance that allows us to analyze real-world systems.

We conclude this journey with an extract of "The Avionics Handbook", which gives a promising perspective to the work described in this Thesis.

> *There are two major reliability factors to be addressed in the design of ultrareliable avionics: hardware component failures and design errors. Physical component failures can be handled by using redundancy and voting. Formal methods address the problem of design errors.*
>
> – Sally C. Johnson and Ricky W. Butler. The Avionics Handbook.

Bibliography

[1] Martín Abadi and Leslie Lamport. Composing Specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, 1993.

[2] Sherif Abdelwahed, Gabor Karsai, Nagabhushan Mahadevan, , and Stanley Ofsthun. Practical Implementation of Diagnosis Systems Using Timed Failure Propagation Graph Models. *IEEE T. Instrumentation and Measurement*, 58(2):240–247, 2009.

[3] Parosh Aziz Abdulla, Johann Deneux, Gunnar Stålmarck, Herman Ågren, and Ove Åkerlund. Designing Safe, Reliable Systems Using Scade. In *Leveraging Applications of Formal Methods, First International Symposium, ISoLA 2004, Paphos, Cyprus, October 30 - November 2, 2004, Revised Selected Papers*, pages 115–129, 2004.

[4] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. A Parametric Approach for Smaller and Better Encodings of Cardinality Constraints. In *Proceedings of CP*, 2013.

[5] Jacob A. Abraham and Daniel P. Siewiorek. An algorithm for the accurate reliability evaluation of triple modular redundancy networks. *IEEE Trans. on Comp.*, 100(7):682–692, 1974.

[6] Jean-Raymond Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge Univ. Press, 1996.

[7] Federal Aviation Administration. Advisory Circular 20-174. http://www.faa.gov/documentLibrary/media/Advisory_Circular/AC%2020-174.pdf.

[8] Federal Aviation Administration. Advisory Circular 23-1309-1E. http://www.faa.gov/documentLibrary/media/Advisory_Circular/AC%2023.1309-1E.pdf.

[9] Federal Aviation Administration. System Design and Analysis Document Information, 2002. FAA Advisory Circular 25.1309-1.

[10] Airbus. A330, Flight Deck and Systems Briefing for Pilots, 1999. STL 472.755/92.

[11] Ove Åkerlund, Pierre Bieber, Eckard Boede, Marco Bozzano, Matthias Bretschneider, Charles Castel, Antonella Cavallo, Massimo Cifaldi, Jean Gauthier, Alain Griffault, et al. ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects. *Proc. ERTS*, 2006, 2006.

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

[13] André Arnold, Gérald Point, Alain Griffault, and Antoine Rauzy. The AltaRica formalism for describing concurrent systems. *Fundamenta Informaticae*, 40:109–124, 2000.

[14] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In *Automated Deduction CADE-18*, pages 195–210. Springer, 2002.

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

[16] Iain Bate, Richard Hawkins, and John A. McDermid. A Contract-based Approach to Designing Safe Systems. In *SCS*, pages 25–36, 2003.

[17] 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.

[18] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Munoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saıdi, Natarajan Shankar, et al. An overview of sal. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*, 2000.

[19] Albert Benveniste, Benoit Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. Multiple Viewpoint Contract-Based Specification and Design. In *FMCO*, pages 200–225, 2007.

[20] Cinzia Bernardeschi, Alessandro Fantechi, Stefania Gnesi, Giorgio Mongardi, and Giorgio. Proving safety properties for embedded control systems, 1996.

[21] Pierre Bieber, Christian Bougnol, Charles Castel, Jean-Pierre Christophe Kehren, Sylvain Metge, and Christel Seguin. Safety assessment with altarica. In *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 505–510. 2004.

[22] Pierre Bieber, Charles Castel, and Christel Seguin. Combination of Fault Tree Analysis and Model Checking for Safety Assessment of Complex System. In *Proc. EDCC-4*, volume 2485 of *LNCS*, pages 19–31. Springer, 2002.

[23] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002.

[24] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proc. TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[25] Armin Biere and Keijo Heljanko. Hardware Model Checking Competition, 2015. http://fmv.jku.at/hwmcc15/.

[26] Armin Biere, Keijo Heljanko, and Siert Wieringa. *AIGER*, 2011. `http://fmv.jku.at/aiger/`.

[27] Benjamin Bittner, Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Marco Gario, Alberto Griggio, Cristian Mattarei, Andrea Micheli, and Gianni Zampedri. The xsap safety analysis platform. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science. Springer Verlag, 2015.

[28] Andrea Bondavalli, Mario Dal Cin, Diego Latella, István Majzik, András Pataricza, and Giancarlo Savoia. Dependability analysis in the early phases of UML-based system design. *Comput. Syst. Sci. Eng.*, 16(5):265–275, 2001.

[29] Marco Bozzano, Antonella Cavallo, Massimo Cifaldi, Laura Valacca, and Adolfo Villafiorita. Improving safety assessment of complex sys-

tems: An industrial case study. In *Proceedings of Formal Methods 2003 (LNCS 2805*, pages 208–222. Springer-Verlag, 2003.

[30] Marco Bozzano, Alessandro Cimatti, Alberto Griggio, and Cristian Mattarei. Efficient anytime techniques for model-based safety analysis. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 603–621, 2015.

[31] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. Safety, dependability, and performance analysis of extended AADL models. *The Computer Journal*, doi: 10.1093/com, March 2010.

[32] Marco Bozzano, Alessandro Cimatti, Oleg Lisagor, Cristian Mattarei, Sergio Mover, Marco Roveri, and Stefano Tonetta. Safety Assessment of AltaRica Models via Symbolic Model Checking. *Science of Computer Programming*, 98(4):464 – 483, 2015.

[33] Marco Bozzano, Alessandro Cimatti, and Cristian Mattarei. Automated analysis of reliability architectures. In *ICECCS*, pages 198–207. IEEE Computer Society, 2013.

[34] Marco Bozzano, Alessandro Cimatti, and Cristian Mattarei. Efficient Analysis of Reliability Architectures via Predicate Abstraction. In *Proc. HVC*, number 8244 in LNCS, pages 279–294. Springer, 2013.

[35] Marco Bozzano, Alessandro Cimatti, Cristian Mattarei, and Stefano Tonetta. Formal Safety Assessment via Contract-Based Design. In *Proc. ATVA*, number 8837 in LNCS, pages 81–97. Springer, 2014.

[36] Marco Bozzano, Alessandro Cimatti, Anthony Fernandes Pires, David Jones, Greg Kimberly, Tyler Petri, Richard Robinson, and

Stefano Tonetta. A formal account of the AIR6110 Wheel Brake System. 2015. Submitted to CAV'15.

[37] Marco Bozzano, Alessandro Cimatti, and Francesco Tapparo. Symbolic fault tree analysis for reactive systems. In *ATVA*, pages 162–176, 2007.

[38] Marco Bozzano and Adolfo Villafiorita. Improving system reliability via model checking: The fsap/nusmv-sa safety analysis platform. In *SAFECOMP*, pages 49–62, 2003.

[39] Marco Bozzano and Adolfo Villafiorita. Integrating Fault Tree Analysis with Event Ordering Information. In *Proceedings of ESREL 2003*, pages 247–254, 2003.

[40] Marco Bozzano and Adolfo Villafiorita. The FSAP/NuSMV-SA Safety Analysis Platform. *Software Tools for Technology Transfer*, 9(1):5–24, 2007.

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

[42] Marco Bozzano, Adolfo Villafiorita, Ove Åkerlund, Pierre Bieber, Christian Bougnol, Eckard Böde, Matthias Bretschneider, Antonella Cavallo, et al. ESACS: an integrated methodology for design and safety analysis of complex systems. *Proc. ESREL 2003*, pages 237–245, 2003.

[43] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In *VMCAI*, pages 70–87, 2011.

[44] Aaron R. Bradley and Zohar Manna. Checking safety by inductive generalization of counterexamples to induction. In *Formal Methods*

in Computer Aided Design, 2007. FMCAD'07, pages 173–180. IEEE, 2007.

[45] Robert Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In *Computer Aided Verification*, pages 24–40. Springer, 2010.

[46] Manfred Broy. Towards a Theory of Architectural Contracts: - Schemes and Patterns of Assumption/Promise Based System Specification. In *Software and Systems Safety - Specification and Verification*, pages 33–87. IOS Press, 2011.

[47] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[48] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and Lain-Jinn Hwang. Symbolic model checking: 10 20 states and beyond. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 428–439. IEEE, 1990.

[49] Kai-Yuan Cai. System failure engineering and fuzzy methodology an introductory overview. *Fuzzy sets and systems*, 83(2):113–133, 1996.

[50] 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 *Proc. CAV*, pages 334–342, 2014.

[51] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic

Model Checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002.

[52] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: a new symbolic model checker. *STTT International Journal on Software Tools for Technology Transfer. Editors-in-Chief: B. Steffen - W. R. Cleaveland. Springer Verlag.*, pages 410–425, 2000.

[53] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: a new Symbolic Model Checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):410–425, March 2000.

[54] Alessandro Cimatti, Michele Dorigatti, and Stefano Tonetta. OCRA: A Tool for Checking the Refinement of Temporal Contracts. In *ASE*, pages 702–705. IEEE, 2013.

[55] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Parameter synthesis with IC3. In *Proceedings of FMCAD*, pages 165–168. IEEE, 2013.

[56] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *TACAS*, pages 93–107, 2013.

[57] Alessandro Cimatti, Marco Roveri, and Stefano Tonetta. Requirements validation for hybrid systems. In *Computer Aided Verification*, pages 188–203. Springer, 2009.

[58] 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.

[59] Alessandro Cimatti and Stefano Tonetta. Contracts-refinement proof system for component-based embedded systems. *Science of Computer Programming*, 97:333–348, 2015.

[60] Koen Claessen and Niklas Sörensson. A liveness checking algorithm that counts. In *FMCAD*, pages 52–59, 2012.

[61] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 321–330. ACM, 2011.

[62] Olivier Coudert and Jean Christophe Madre. A new method to compute prime and essential prime implicants of boolean functions. *Advanced Research in VLSI and Parallel Systems, Knight and Savage (Eds)*, pages 113–128, 1992.

[63] Olivier Coudert and Jean Christophe Madre. Fault Tree Analysis: 10^{20} Prime Implicants and Beyond. In *Proc. RAMS*, 1993.

[64] Werner Damm, Hardi Hungar, Bernhard Josko, Thomas Peikenkamp, and Ingo Stierand. Using contract-based component specifications for virtual integration testing and architecture design. In *DATE*, pages 1023–1028, 2011.

[65] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003.

[66] E. Allen Emerson and Chin-Laung Lei. Temporal model checking under generalized fairness constraints. In *Proceedings of the 18th Annual Hawaii International Conference on System Sciences*, pages 84–96, 1985.

[67] Christian Favre. Fly-by-wire for commercial aircraft: the airbus experience. *International Journal of Control*, 59(1):139–157, 1994.

[68] Peter Fenelon, John A. McDermid, Mark Nicolson, and David J. Pumfrey. Towards integrated safety analysis and design. *ACM SIGAPP Applied Computing Review*, 2(1):21–32, 1994.

[69] Marco Gario, Alessandro Cimatti, Cristian Mattarei, Stefano Tonetta, and Kristin Y. Rozier. Model checking at scale: Automated air traffic control design space exploration. In *Under Submission*.

[70] Marco Gario and Andrea Micheli. PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In *SMT-Workshop*, 2015.

[71] Nouredine Hadjsaid, Marie-Cecile Alvarez-Herault, Raphael Caire, Bertrand Raison, Justine Descloux, and Wojciech Bienia. Novel architectures and operation modes of distribution network to increase dg integration. In *IEEE conference, General Meeting 2010, Minneapolis, USA*. IEEE, 2010.

[72] 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.

[73] Gerard J. Holzmann. The model checker SPIN. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.

[74] Husni R. Idris, Ni Shen, and David J. Wing. Improving separation assurance stability through trajectory flexibility preservation. In *10th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference*, page 9011, 2010.

[75] Frantz Iwu, Andy Galloway, John A. McDermid, and Ian Toyn. Integrating safety and formal analyses using UML and PFS. *Reliability Engineering & System Safety*, 92(2):156–170, 2007.

[76] Jean-Baptiste Jeannin, Khalil Ghorbal, Yanni Kouskoulas, Ryan Gardner, Aurora Schmidt, Erik Zawadzki, and André Platzer. A formally verified hybrid system for the next-generation airborne collision avoidance system. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 21–36. Springer, 2015.

[77] Jonathan M. Johnson and Michael 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.

[78] Kamiar Karimi. Future aircraft power systems- integration challenges. In *CMU Conference on the Electricity Industry, Feb. 2008, Pittsburg, PA.* CMU, 2008.

[79] David A. Karr, Robert A. Vivona, David A. Roscoe, Stephen M. DePascale, and David J. Wing. Autonomous operations planner: A flexible platform for research in flight-deck support for airborne self-separation. In *12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference and 14th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, page 5417, 2012.

[80] Farid Katiraei. Novel microgrids management: Controls and operation aspects of microgrids. *IEEE Power Energy Magazine*, (May/June), 2008.

[81] Andrew J. Kornecki and Kimberley Hall. Approaches to assure safety in fly-by-wire systems: Airbus vs. boeing. In *IASTED Conf. on Software Engineering and Applications*, pages 471–476, 2004.

[82] 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.

[83] Donald C. Latham. Department of defense trusted computer system evaluation criteria. *Department of Defense*, 1986.

[84] Todd Lauderdale, Timothy Lewis, Thomas Prevot, Mark Ballin, Arwa Aweiss, and Nelson Guerreiro. Function allocation for separation assurance: Research plan. NASA HQ Project Overview, Aug. 2014.

[85] Christoph Lauer, Reinhard German, and Jens Pollmer. Fault tree synthesis from uml models for reliability analysis at early design stages. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011.

[86] LayerZero Power Systems, Inc. – Triple Modular Redundancy. `http://www.layerzero.com/Innovations/Industry-Firsts/Triple-Modular-Redundancy.html`.

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

[88] Panagiotis Manolios, Daron Vroon, and Gayatri Subramanian. Automating component-based system assembly. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 61–72. ACM, 2007.

[89] Evelyn Mathison and Kamiar Karimi. Power quality specification development for more electric airplane architectures. In *Power System*

Conference Proceedings, October 2002, number SAE 2002-01-3266. SAE, October 2002.

[90] Cristian Mattarei, Alessandro Cimatti, Marco Gario, Stefano Tonetta, and Kristin Y. Rozier. Comparing different functional allocations in automated air traffic control design. In *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*, pages 112–119, 2015.

[91] Mark L. McKelvin Jr, Gabriel Eirea, Claudio Pinello, Sri Kanajan, and Alberto Sangiovanni-Vincentelli. A formal approach to fault tree synthesis for the analysis of distributed fault tolerant systems. In *EMSOFT*, pages 237–246. ACM, 2005.

[92] Kenneth L. McMillan. *Symbolic model checking*. Springer, 1993.

[93] Kenneth L. McMillan. Interpolation and sat-based model checking. In *CAV*, pages 1–13, 2003.

[94] The MISSA Project. `http://www.missa-fp7.eu`.

[95] UK MoD. The procurement of safety critical software in defence equipment. *Interim Defence Standard 00-55, UK Ministry of Defence, Directorate of Standardization, Kentigern House*, 65, 1991.

[96] Ian Moir and Allan Seabridge. *Aircraft Systems: Mechanical, Electrical and Avionics Subsystems Integration*. John Wiley & Sons, 2008.

[97] nuXmv: a new eXtended model verifier. `https://nuxmv.fbk.eu`.

[98] OpenFTA: OpenFTA is an advanced tool for fault tree analysis. `http://www.openfta.com/`.

[99] Ganesh J. Pai and Joanne Bechta Dugan. Automatic synthesis of dynamic fault trees from UML system models. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, pages 243–254. IEEE, 2002.

[100] Yiannis Papadopoulos and John A. McDermid. Hierarchically performed hazard origin and propagation studies. In *in Lecture Notes in Computer Science, 1698:139-152, Proceedings of SAFECOMP'99, the 18 th International Conference on Computer Safety, Reliability and Security*, pages 139–152. Springer Verlag, 1999.

[101] Claudio Pinello, Luca Carloni, and Alberto Sangiovanni-Vincentelli. Fault-tolerant deployment of embedded software for cost-sensitive real-time feedback-control applications. In *DATE*, page 21164. IEEE Computer Society, 2004.

[102] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[103] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. *International Journal on Software Tools for Technology Transfer*, 7(2):156–173, 2005.

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

[105] Antoine Rauzy. Mathematical Foundations of Minimal Cutsets. *IEEE Transactions on Reliability*, 50(4):389–396, 2001.

[106] Anish Sachdeva, Dinesh Kumar, and Pradeep Kumar. Reliability modeling of an industrial system with petri nets. In *Proceedings of ESREL*, pages 1087–94, 2007.

[107] SAE. ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, dec 1996.

[108] SAE. ARP4754A Guidelines Guidelines for Development of Civil Aircraft and Systems, dec 2010.

[109] SAE. AIR 6110, Contiguous Aircraft/ System Development Process Example, December 2011.

[110] Manel Sghairi, Agnan De Bonneval, Yves Crouzet, Jean-Jacques Aubert, and Patrice Brot. Challenges in building fault-tolerant flight control system for a civil aircraft. *IAENG International Journal of Computer Science*, 35(4), 2008.

[111] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *FMCAD*, pages 108–125, 2000.

[112] Sajjad Siddiqi and Jinbo Huang. Hierarchical Diagnosis of Multiple Faults. In *IJCAI*, pages 581–586, 2007.

[113] Ivan Sikora, Stanislav Pavlin, and Ernest Bazijanac. Flight operations and engineering documentation managing and distribution supported by intelligent transport systems. 2000.

[114] Cary R. Spitzer. *The Avionics Handbook*. CRC Press, 2000.

[115] Darshan D. Thaker, Rajeevan Amirtharajah, Francois Impens, Isaac L. Chuang, and Frederic T. Chong. Recursive TMR: Scaling fault tolerance in the nanoscale era. *Design & Test of Computers, IEEE*, 22(4):298–305, 2005.

[116] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *1st Symposium in Logic in Computer Science (LICS)*. IEEE Computer Society, 1986.

[117] William E. Vesely, Joanne Bechta Dugan, Joseph Fragola, Joseph Minarick, and Jan Railsback. Fault tree handbook with aerospace applications. *NASA Office of Safety and Mission Assurance*, 2002.

[118] William E. Vesely, Francine F. Goldberg, Norman H. Roberts, and David F. Haasl. Fault tree handbook. Technical Report NUREG-0492, Systems and Reliability Research Office of Nuclear Regulatory Research U.S. Nuclear Regulatory Commission, 1981.

[119] Christian von Essen and Dimitra Giannakopoulou. Analyzing the next generation airborne collision avoidance system. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 620–635. Springer, 2014.

[120] David J. Wing, Mark G. Ballin, and Karthik Krishnamurthy. Pilot in command: a feasibility assessment of autonomous flight management operations. In *24th International Congress of the Aeronautical Sciences*, 2004.

[121] Xilinx. TMRTool.

[122] Ying C. Yeh. Triple-Triple Redundant 777 Primary Flight Computer. In *Aerospace Applications Conference, 1996. Proc., IEEE*, volume 1, pages 293–307. IEEE, 1996.

[123] Miaomiao Zhang, Zhiming Liu, Charles Morisset, and Anders Ravn. Design and verification of fault-tolerant components. *Methods, Models and Tools for Fault Tolerance*, pages 57–84, 2009.

[124] Yang Zhao and Kristin Yvonne Rozier. Formal specification and verification of a coordination protocol for an automated air traffic control system. *Science of Computer Programming Journal*, 96(3):337–353, December 2014.

[125] Yang Zhao and Kristin Yvonne Rozier. Probabilistic model checking for comparative analysis of automated air traffic control systems. In *Proceedings of the 33rd IEEE/ACM International Conference On Computer-Aided Design (ICCAD 2014)*, page To appear, San Jose, California, U.S.A., November 2014. IEEE/ACM.

[126] Enrico Zio. Reliability engineering: Old problems and new challenges. *Reliability Engineering & System Safety*, 94(2):125–141, 2009.

[127] Enrico Zio and Marzio Marseguerra. Basics of the monte carlo method with application to system reliability. *LiLoLe, Hagen*, 2002.