



UNIVERSITY OF TRENTO, ITALY
DEPARTMENT OF
INFORMATION ENGINEERING AND COMPUTER SCIENCE

Doctoral Thesis

**Automatic Design Space Exploration of
Fault-tolerant Embedded Systems Architectures**

Author: **Antonio Tierno**

Supervisors: **Roberto Passerone, Alessandro Cimatti**

January, 2023

To my family

Acknowledgements

First and foremost I am extremely grateful to my supervisors Roberto Passerone and Alessandro Cimatti for all their help, invaluable advice, patience, and support during my PhD study. Their immense knowledge and plentiful experience have encouraged me in all the time of my academic research and daily life. Furthermore, I would also like to thank Giuliano Turri for his help to make significant progress in the thesis work and for technical support. In addition, I would like to thank all the members in the Department of Information Engineering and Computer Science of the ICT International Doctoral School of the University of Trento for their kind help and constant support. Finally, I would like to express my gratitude to my parents, my wife, and my son. Without their patience, understanding, and encouragement, it would be impossible for me to complete my study.

Abstract

Embedded Systems may have competing design objectives, such as to maximize the reliability, increase the functional safety, minimize the product cost, and minimize the energy consumption. The architectures must be therefore configured to meet varied requirements and multiple design objectives. In particular, reliability and safety are receiving increasing attention. Consequently, the configuration of fault-tolerant mechanisms is a critical design decision. This work proposes a method for automatic selection of appropriate fault-tolerant design patterns, optimizing simultaneously multiple objective functions. Firstly, we present an exact method that leverages the power of Satisfiability Modulo Theory to encode the problem with a symbolic technique. It is based on a novel assessment of reliability which is part of the evaluation of alternative designs. Afterwards, we empirically evaluate the performance of a near-optimal approximation variation that allows us to solve the problem even when the instance size makes it intractable in terms of computing resources. The efficiency and scalability of this method is validated with a series of experiments of different sizes and characteristics, and by comparing it with existing methods on a test problem that is widely used in the reliability optimization literature.

Keywords: Embedded systems, fault-tolerance, reliability analysis, redundancy allocation, design space exploration, automatic synthesis, satisfiability theory, multiple-objective optimization.

Contents

List of Figures	xi
List of Tables	xix
Listings	xxi
List of Acronyms	xxiii
1 Introduction	1
1.1 Context	1
1.2 Problem Statement and Motivation	2
1.3 Research Questions and Contribution	2
1.4 Outline	3
2 Background Notions	5
2.1 Reliability Assurance of Complex Systems	5
2.1.1 Basic Definitions	6
2.1.2 Basic Concepts	6
2.1.3 RAMS Standards	8
2.1.4 Common Functions for Modeling Reliability	9
Probability Distributions	10
2.1.5 Failure Classification	12
2.1.6 Failure Modes Analysis	14
2.1.7 Concept of Redundancy	19
2.1.8 Design Patterns for Reliability	20
Hardware Patterns	20
Software Patterns	20
Hardware and Software Patterns	21
2.1.9 Dealing with System Faults	21
2.1.10 Dealing with Non-Functional Requirements	22
2.2 Architecture-based reliability evaluation	22

2.2.1	Combinatorial Models	23
	Reliability Block Diagrams	23
	Fault Tree Analysis	25
	Binary Decision Diagram	26
2.2.2	State-based Models	31
	Markov Models	32
	Petri Net Models	33
2.2.3	Simulation-based Approaches	35
2.3	Formal Methods, Techniques, and Tools	35
2.3.1	Computational Models	36
	Methods	36
	Modeling Languages	38
	Architectural Modeling	38
2.3.2	Formal Methods for Specification	41
2.3.3	Formal Methods for Verification	41
	Hints of Boolean Algebra	42
	Logic in a Nutshell	43
	Proof Tools	47
	Model Checking	48
2.3.4	Formal Methods for Implementation	49
2.3.5	How to Choose a Formal Method	49
	System classification	50
	System properties	51
	Summary	51
2.4	System Optimization	52
2.4.1	Single-objective Optimization	52
	Branch and Bound	52
	Mathematical Programming	53
	Local Search	54
2.4.2	Multi-objective Optimization	56
	Linear Programming	57
	Meta-heuristic Search Algorithms	58
	Answer Set Programming	59
	Satisfiability	59
	Satisfiability Modulo Theories	59
	Optimization Modulo Theories	60
2.4.3	Approaches to Design Space Exploration	60

3	Related Work	63
3.1	Approaches to the Design and Analysis of Fault-Tolerant Systems	63
3.1.1	Design for Graceful Degradation	64
3.1.2	Design for Robustness	68
3.1.3	Design for Mixed Criticality	69
3.1.4	Design for Reconfiguration	71
3.1.5	Design for Self-x	72
3.2	Formal Reliability Analysis of Redundant Architectures	73
3.3	High Level Synthesis Optimization	75
3.3.1	Configuration or System Assembly Problems	76
3.3.2	Redundancy Allocation Problems	79
3.3.3	Selection Problems	81
3.3.4	Placement Problems	83
3.3.5	Routing Problems	84
3.3.6	Deployment Optimization Problems	86
3.3.7	Resource allocation Problems	87
3.3.8	Scheduling and Sequencing Problems	90
3.3.9	Workflow Satisfiability Problems	92
3.4	Approaches to Automatic Verification	93
3.4.1	Automated Theorem Proving	94
3.4.2	Symbolic Model Checking	94
3.4.3	Bounded Model Checking	96
3.4.4	SMT Model Checking	97
3.4.5	OMT Model Checking	98
3.4.6	Equivalence Checking	98
3.4.7	Static Analysis	99
3.4.8	Semiformal Verification	100
3.4.9	Conclusion	100
4	Proposed Method	103
4.1	Research Methodology	103
4.2	Research Problem	104
4.2.1	Existing Limitations	104
4.2.2	Problem Formulation	105
4.3	Overview of the Approach	106
4.4	Input and Output	107
4.4.1	System Model	108

4.4.2	Fault Model	108
4.4.3	Objectives	109
4.4.4	Redundant Patterns	111
4.4.5	Design Constraints	111
4.5	Contributions	111
4.6	Challenges	113
5	Reliability Assessment of Redundant Architectures	117
5.1	Assumptions	117
5.2	Modeling the System Architecture	118
5.3	Modeling the Miter	124
5.4	Minimal Cut-sets Computation	126
5.5	Reliability Assessment	128
5.6	Improvements and Refinements	129
5.6.1	Minimal Cut Sets Computation via Predicate Abstraction	129
5.6.2	Reducing the Number of Decision Variables	132
5.6.3	Management of Uncertain Cases	132
5.6.4	Caching Mechanism	134
5.7	Work Extensions	134
6	Design Space Exploration of Redundant Architectures	137
6.1	DSE Features	137
6.1.1	Design Space Representation	138
6.1.2	Design Space Generation	138
6.1.3	Exploration Method	138
6.1.4	Evaluation	139
6.1.5	Selection	139
6.1.6	Refinement	139
6.2	Constraint Solving Approach	140
6.2.1	Formalization of Constraints	140
6.3	Problem Encoding	142
6.3.1	Modeling the System Architecture	144
6.3.2	Construction of a Library of Redundant Patterns	145
6.3.3	Fault Model	150
6.3.4	Modeling the Redundant Architecture	151
6.3.5	Generation of All Redundant Configurations.	152
6.3.6	Modeling the Miter	158

6.3.7	Reliability assessment	160
	Explicit Representation	161
	Symbolic Representation	162
	Semi-symbolic Representation	169
6.3.8	Assessment of Other Non-functional Parameters	169
6.3.9	Optimization	170
6.3.10	Improvements and Refinements	170
	Minimal Cut-Sets Computation of Symbolic representa- tions	170
	Choosing Optimal Variable ordering	174
	Using Binary Encoding to Encode Configuration Variables	182
7	Experimental Evaluation of Exact Method	187
7.1	Implementation Framework	187
7.2	Implementation Details	188
7.3	Running Example	191
	Explicit Method	192
	Symbolic Method	192
	Semi-symbolic Method	193
7.3.1	Varying the Number of Objective Functions	194
7.3.2	Varying the Number of Redundant Patterns	202
7.3.3	Varying the Number of System Components	202
7.3.4	Varying the BDD Ordering Strategy	202
7.4	Benchmarks	203
	7.4.1 Experimental Setup	204
	7.4.2 Evaluation Criteria	205
	7.4.3 Experiments on Linear Architectures	205
	7.4.4 Experiments on Rectangular Architectures	206
	7.4.5 Experiments on Complex Architectures	207
7.5	Results	208
	7.5.1 Assessment Performance	209
	Task 1: Abstraction	209
	Task 2: BDD-based Quantifier Elimination	211
	Task 3: BDD Traversing	213
	7.5.2 Optimization Performance	215
7.6	Test problem	218
7.7	Applicability and Limitations	219

8	Near-Optimal Approximations	221
8.1	Simplifying the Exact Method	221
8.2	Graph Partitioning	222
8.2.1	Kernighan–Lin Algorithm	222
8.2.2	Multi-level Partitioning	223
8.3	Partitioning the System Architecture	225
8.4	Combining Solutions from Sub-architectures	225
8.5	Pruning and Ranking for Large Problems	225
8.6	Running Example	229
9	Experimental Evaluation of Approximate Method	235
9.1	Implementation details	235
9.2	Benchmarks	236
9.2.1	Experimental setup	236
9.2.2	Evaluation criteria	236
9.2.3	Experiments on complex architectures	238
	Example system with 8 components	238
	Example system with 10 components	239
	Example system with 14 components	242
	Example system with 24 components	243
	Example system with 69 components	246
	Strategy to determine the pruning threshold	247
9.3	Results	248
9.4	Test problem	249
9.5	Applicability and Limitations	259
10	Conclusions and Future Work	261
10.1	Summary	261
10.2	Models Assumptions, Limitations, and Applicability	262
10.3	Exact or Approximate Method?	263
10.4	Application to real systems	264
10.5	Remarks	266
10.6	Future work	267
	References	268
	Appendix A Software Dependency Graph	305
A.1	Software structure	305

Appendix B Installation of Required Tools	309
B.1 Software needed	309
B.2 Installation Procedure	310
B.2.1 Installing Python	310
B.2.2 Installing Python IDE [OPT]	311
B.2.3 Installing pySMT	311
B.2.4 Installing Solvers	311
B.2.5 Installing other useful packages	314

List of Figures

1.1	Redundant system-level synthesis flow	3
2.1	Safety engineering lifecycle from Trapp et Al. [6].	8
2.2	Safety regulations, norms, and standards.	9
2.3	FTA symbols	15
2.4	Example of fault tree diagram	16
2.5	Examples of conversion from FTD to RBD	17
2.6	RBD equivalent to fault tree of Figure 2.4	18
2.7	Reliability block diagrams for series (a), parallel (b), and k-out-of-n (c) systems.	24
2.8	Complex system composed by seven components	25
2.9	Reliability block diagram for system in Figure 2.8	25
2.10	Fault trees for series (a), parallel (b), and k-out-of-n (c) systems.	26
2.11	Fault tree for system in Figure 2.8	27
2.12	Binary tree of the example formula $F = a \wedge b + \neg a \wedge c$	28
2.13	Binary tree reduction rules: (a) Elimination, (b) Isomorphism.	28
2.14	BDD of the example formula $F = a \wedge b + \neg a \wedge c$	29
2.15	Example of a series system in the form of a <i>Binary Decision Diagram</i> (BDD)	30
2.16	Example of a parallel system in the form of a BDD	30
2.17	Example of a k-o-o-n system in the form of a BDD, with k=3 and n=5.	31
2.18	General Markov model of a TMR system	33
2.19	Petri Net model of a TMR system	34
2.20	Resolution methods of <i>Multi-Objective Optimization Problem</i> (MOOP)s	57
3.1	System assembly problem as part of the system development life cycle	77

3.2	Redundancy allocation problem as part of the system development lifecycle	79
3.3	Selection problem as part of the system development lifecycle . .	82
3.4	Placing and routing problems as part of the system development lifecycle	85
3.5	Mapping and scheduling problem	88
3.6	Workflow Satisfiability problem	92
3.7	Steps performed by NuSMV2 [270].	95
4.1	Redundant system-level synthesis flow	107
4.2	Inputs and outputs of <i>Design Space Exploration</i> (DSE) approach proposed	108
5.1	Example of series (a), parallel (b), and complex (c) system architectures.	119
5.2	Example module with nominal and faulty behavior	120
5.3	Example of redundant pattern modeled with nominal and faulty behavior	121
5.4	Output of the test example for TMR	123
5.5	Linking constraints determine the connections between two components	124
5.6	Miter composition	125
5.7	A stage aggregates nominal and faulty behaviors	126
5.8	Stage-based Miter composition	127
5.9	OBDD of the formula	129
5.10	Abstract stage (aka CSA)	130
5.11	Abstract miter	131
5.12	Abstract miter is composed by abstract stages	131
5.13	Reduced CSA for a TMR example pattern	132
5.14	Caching improves the performance by storing patterns behavior formulae and accessing them on later requests.	135
6.1	The proposed method consists of three main phases: modeling of redundant system, assessment of non-functional parameters, and optimization.	143
6.2	Multi-objective DSE flow	145
6.3	Comparator design pattern	146
6.4	Duplex design pattern	147

6.5	TMR design pattern (<i>TMR_V111</i>)	147
6.6	Different <i>Triple Modular Redundancy</i> (TMR) configurations . .	148
6.7	M-o-o-N design pattern	149
6.8	Sparing design pattern	149
6.9	An example of extended component	150
6.10	An example of extended component	151
6.11	Extended TMR	151
6.12	Example of basic (non-redundant) architecture composed of three components connected in series and a library of seven redundant design patterns.	154
6.13	Fault atoms for patterns P_1 (a), P_2 (b), and P_3 (c).	155
6.14	Configuration and fault variables	155
6.15	Set of redundant architectures for example system	157
6.16	Redundant alternatives for system in Figure 6.12	157
6.17	Miter composition for architecture of Figure 6.16a	158
6.18	Stage-based Miter for architecture of Figure 6.16a	158
6.19	Abstract Miter composition for architecture of Figure 6.16a . . .	159
6.20	Redundant architecture alternative 1 of running example	162
6.21	Fault variables for redundant architecture alternative 1	165
6.22	Excerpt of an OBDD encoding the <i>Cut-Set</i> (CS)s of the running example	166
6.23	Examples of BDD traversing: $cfg_1 = \top$ and $cfg_2 = \top$ (a), $cfg_1 = \top$ and $cfg_2 = \perp$ (b), $cfg_1 = \perp$ and $cfg_2 = \perp$ (c).	168
6.24	Example of basic (non-redundant) system and a library includ- ing instances of the same pattern type.	171
6.25	Combinatorial abstract Miter composition for architecture of Figure 6.24	173
6.26	Example system of two components, each with one two redun- dant pattern.	175
6.27	Partial BDD for example in Figure 6.26, using ordering with all configuration variables on top.	176
6.28	BDD for example in Figure 6.26, using arbitrary ordering of variables.	177
6.29	BDD for example in Figure 6.26, using an alternative arbitrary ordering of variables.	178
6.30	BDD for example in Figure 6.26, using an ordering that follows architecture's topology.	179

6.31	Variable orderings used for the BDD construction of example in Figure 6.26	180
6.32	BDD of example in Figure 6.26 resulting from different orderings: all variables on top (a), arbitrary assignment (b), alternative arbitrary assignment(c), driven by architecture’s topology (d).	185
6.33	BDD of example in Figure 6.26 using binary encoding of configuration variables and different orderings: all variables on top (a), arbitrary assignment (b), alternative arbitrary assignment(c), driven by architecture’s topology (d).	186
7.1	Conversion from PySMT formula to <i>Canonical Complementary Edge</i> (CCE)-BDD representation using CUDD.	190
7.2	Basic system of first example.	192
7.3	Pareto solutions for the example system in Figure 7.2	194
7.4	Alternative solutions for example system in Figure 7.2: (a) solution n.6, (b) redundant architecture scheme with minimum cost, and (c) redundant architecture scheme with maximum reliability.199	
7.5	Pareto surface for example system in Figure 7.2 with three objective functions.	200
7.6	Time performance for optimization when varying the number of objectives for example system in Figure 7.2	200
7.7	Memory usage for optimization when varying the number of objectives for example system in Figure 7.2	200
7.8	Parallel coordinate plot for the running example with six objective functions	201
7.9	System architectures used fer experimental evaluation: (a) series (aka linear) (b) repeating pairs of parallel components (aka rectangular), (c) complex random architectures.	204
7.10	Time performance for linear architectures, varying the size of the system and library of patterns.	206
7.11	Time performance for rectangular architectures, varying the size of the system and library of patterns.	206
7.12	Pareto solutions for a basic system composed of 8 components and 10 edges.	207
7.13	Pareto solutions for a basic system composed of 10 components and 10 edges.	207

7.14	Pareto solutions for a basic system composed of 12 components and 16 edges.	208
7.15	Pareto solutions for a basic system composed of 14 components and 20 edges.	208
7.16	Time and memory performance of exact method varying the number of basic components.	209
7.17	Time performance for extraction of non-functional parameters using a library of different redundant patterns for serial architecture of 100 components.	211
7.18	Time performance for extraction of non-functional parameters using a library of different redundant patterns for rectangular architecture of 100 components, organized in 50 levels.	212
7.19	Time performance of BDD-based quantifier elimination using different redundant patterns, for linear and rectangular architectures.	213
7.20	Time performance of BDD-based quantifier elimination using different instances of the same redundant patterns, for linear and rectangular architectures.	213
7.21	Time performance of different types of dynamic reordering strategies for linear and rectangular architectures.	214
7.22	Time performance of reliability function extraction for linear and rectangular architectures, using different types of patterns.	214
7.23	Time performance for BDD-based quantifier elimination and reliability function extraction for complex architectures.	215
7.24	Time performance for reliability function extraction of complex architectures (d is the maximum incoming degree of the components, i.e., the maximum number of incoming connections).	216
7.25	Number of BDD nodes, time performance of BDD-based quantifier elimination, and reliability function extraction for complex architectures varying the number of components.	217
7.26	Performance of optimization when varying the number of objectives (system of 6 components with 3 patterns each).	217
7.27	Topology used in the Test Problem, with 223 nodes and 252 edges.	218
8.1	(a) A toy example with 6 vertices and seven edges, (b) cyclic 2-way partitioning (c) acyclic partitioning of the same directed graph.	224

8.2	Approximate method for DSE of redundant architectures, overall process flow.	226
8.3	The pruning strategy allows us to adapt the search area by varying a threshold for solution acceptance: all the solutions above the threshold are discarded. This is a flexible solution that can be tuned on the basis of problem instance size and resources available.	227
8.4	Three different partitions for the running example, using KL algorithm.	229
8.5	Applying the pruning strategy to the two partitions of our running example lead to the selection of four solutions for the first partition and one solution for the second one.	231
8.6	Comparison of approximate solutions for the three partitionings in Figure 8.4 without pruning.	233
8.7	Comparison of approximate solutions for the three partitionings in Figure 8.4 with pruning.	233
9.1	The <i>Kernighan–Lin</i> (KL) algorithm included in the NetworkX package.	236
9.2	Source code for the KL algorithm employed.	237
9.3	(a) Basic system composed of 8 components and 10 edges, (b) Pareto solutions of exact and approximate methods.	238
9.4	Comparison of performance of exact and approximate methods for a complex system of eight components.	239
9.5	(a) Partitioning of a basic system composed of 10 components and 13 edges, (b) Pareto solutions of exact and approximate methods.	241
9.6	Comparison of performance of exact and approximate methods for a complex system of ten components.	241
9.7	(a) Partitioning of a basic system composed of 14 components and 20 edges, (b) Pareto solutions of exact and approximate methods.	242
9.8	(a) Comparison of performance of exact and approximate methods for a complex system of fourteen components, (b) Approximate solutions move away from exact solutions as the pruning threshold is tighten up ($th_1 < th_2 < th_3$).	243

9.9	Partitioning of a system composed of 24 components and 33 edges using KL.	244
9.10	Partitioning of a system composed of 24 components and 33 edges using Metis.	244
9.11	Comparison of KL (2 parts) and Metis (3 parts) algorithms for partitioning the example system of 24 components: the two algorithms led to the very same solutions.	245
9.12	For large systems, partitioning in more sub-architectures improves time performance.	245
9.13	Partitioning of a system composed of 69 components and 120 edges using Metis (Number of parts = 3, Edge-cut = 17).	246
9.14	Partitioning of a system composed of 69 components and 120 edges using Metis (Number of parts = 5, Edge-cut = 41).	246
9.15	Pruning of sub-solutions of Part 1 (a) and Part 2 (b) for two different thresholds.	247
9.16	Comparison of time performance of solution methods by varying the number of nodes (a) and the number of edges (b). Note: there is a bump in the curve of approximate solutions because for systems composed by 14 components onwards we employed graph partitioning.	249
9.17	Structure of the <i>Series-Parallel</i> (S-P) system of the test problem	251
9.18	Each stage of the redundant architecture is composed by one to four identical modules in parallel	252
9.19	With our formalism, we used CMP, TMR, and <i>M-Out-Of-N</i> (M-oo-N) patterns to face the test problem.	252
9.20	Component mixing is not allowed in the test problem, i.e., we have to use identical modules for each stage of the architecture.	252
9.21	Partitioning for the test problem system	252
9.22	Solutions for the test problem system. In red, the one with highest reliability.	253
9.23	Comparison of solutions with related works	254
9.24	Solutions for the test problem system. In red, the solution with highest reliability allowing component mixing, in orange the solution obtained without component mixing, illustrated in Figure 9.22.	255

9.25	Comparison of solutions with related works. In red, the solution found by our method with component mixing and applying partitioning with a balanced pruning strategy.	255
9.26	Solutions with higher reliability found by our method.	256
9.27	Comparison of solutions with related works. In purple, the best solution found by our method, by applying partitioning with a pruning strategy designed to favor sub-solutions with higher reliability and lower cost, specific for each sub-architecture. . . .	257
9.28	Comparison of our solution with related works: (a) without component mixing, (b) with component mixing	258
9.29	Example of two architectures producing the same partitionings, without (a) and with (b) cyclic inter-dependencies among components, resulting in good (c) and bad (d) approximate solutions.	259
9.30	The sub-architecture in red includes components not connected in the original architecture.	260
10.1	Power distribution network of an aircraft.	265
A.1	Dependency graph of the software tool implementing the proposed method	307
B.1	Installing Python	310
B.2	Installing Pycharm	311
B.3	Installing PySMT	312
B.4	Adding PySMT to Path	312
B.5	Installing a virtual environment via PyCharm IDE.	313
B.6	Installed solver for pySMT	314
B.7	Installation of CMake	315

List of Tables

2.1	Examples of design methods	37
2.2	Common types of formal models, usually referred to as model of computations.	39
2.3	Examples of architectural languages	40
2.4	Truth table	42
3.1	Configuration (or system assembly) problems	78
3.2	Redundancy allocation problems	81
3.3	Selection problems	83
3.4	Placement and routing problems	86
3.5	Resource allocation problems	90
3.6	Scheduling and Sequencing problems	91
3.7	Workflow satisfiability problems	93
5.1	Time and memory performance for the complex system of the running example illustrated in Figure 7.2	136
6.1	Relations	140
6.2	Library of patterns for example system in Figure 6.12	155
6.3	Binary encoding of (C_i, P_j) allocations	156
6.4	Library of patterns for example system in Figure 6.12	171
7.1	Library of patterns for example system in Figure 7.2, 2 objective functions.	193
7.2	Exact solutions for example system in Figure 7.2, 2 objective functions.	193
7.3	Library of patterns for example system in Figure 7.2, consider- ing 3 objective functions.	195
7.4	Solutions for example system in Figure 7.2, considering 3 objec- tive functions.	195

7.5	Library of patterns for example system in Figure 7.2, considering 4 objective functions.	196
7.6	Solutions for example system in Figure 7.2, considering 4 objective functions.	196
7.7	Library of patterns for example system in Figure 7.2, considering 5 objective functions.	197
7.8	Solutions for example system in Figure 7.2, considering 5 objective functions.	197
7.9	Library of patterns for example system in Figure 7.2, considering 6 objective functions.	198
7.10	Solutions for example system in Figure 7.2, considering 6 objective functions.	198
7.11	Time and memory performance for the complex system of six basic components illustrated in Figure 7.2	202
7.12	Performance for the complex system examples presented above.	208
7.13	Performance of quantifier elimination of the <i>Concretizer-Stage-Abstractor</i> (CSA) of some redundant patterns.	210
7.14	Number of nodes of the <i>Ordered Binary Decision Diagram</i> (OBDD) by varying the ordering strategy. Ordering 1: all configuration variables on top of the OBDD, Ordering 3: driven by architecture.	212
7.15	Optimization: symbolic vs hybrid approach.	216
7.16	Results of the method proposed by Beccuti et al. [308]	218
8.1	Approximate solutions for system in Figure 8.4a (considering two objective functions).	230
8.2	Solutions of the two partitions of system in Figure 8.4a	230
8.3	Solutions of the two partitions of system in Figure 8.4a after pruning	231
8.4	Approximate solutions for example system in Figure 8.4a	231
8.5	Comparison of solutions for system in Figure 8.4a (considering two objective functions)	232
9.1	Comparison of solutions for system in Figure 8.4a (considering two objective functions).	240
9.2	Input data for the test problem	250
9.3	Sub-solutions for the test problem.	253

Listings

5.1	TMR pattern definition	122
5.2	Voter used in the extended TMR	123
5.3	Test example for TMR	123
6.1	AllSMT computation	164
6.2	Extraction of reliability formula (excerpt)	167

List of Acronyms

<i>NLR</i>	<i>Non-Linear Arithmetic over the reals</i>
3-LSM	<i>3-Level Safety Monitoring</i>
AADL	<i>Architecture Analysis and Design Language</i>
ACO	<i>Ant Colony Optimization</i>
ADL	<i>Architecture Description Language</i>
AHGA	<i>Adaptive Hybrid Genetic Algorithm</i>
API	<i>Application Programming Interface</i>
ASP	<i>Answer set programming</i>
AST	<i>Abstract Syntax Tree</i>
AUTOSAR	<i>AUTomotive Open System ARchitecture</i>
AV	<i>Acceptance Voting</i>
BDD	<i>Binary Decision Diagram</i>
BDT	<i>Binary Decision Tree</i>
BFS	<i>Breadth First Search</i>
BMC	<i>Bounded Model Checking</i>
BMS	<i>Building Management Systems</i>
CBS	<i>Constant Bandwidth Server</i>
CCE	<i>Canonical Complementary Edge</i>
CDF	<i>Cumulative Distribution Function</i>
CNF	<i>Conjunctive Normal Form</i>

COP	<i>Constraint Optimization Problem</i>
COTS	<i>Commercial Off-The-Shelf</i>
CP	<i>Constraint Programming</i>
CPS	<i>Cyber-Physical Systems</i>
CS	<i>Cut-Set</i>
CSA	<i>Concretizer-Stage-Abstractor</i>
CSP	<i>Communicating Sequential Processes</i>
CSP	<i>Constraint Satisfaction Problem</i>
CSR	<i>Communicating Shared Resources</i>
CTL	<i>Computation tree logic</i>
CTMC	<i>Continuous Time Markov Chains</i>
DAG	<i>Directed Acyclic Graph</i>
DARTS	<i>Design Approach for Real Time Systems</i>
DFG	<i>Data Flow Graph</i>
DFS	<i>Depth-First Search</i>
DM	<i>Decision Maker</i>
DMR	<i>Deadline missed ratio</i>
DNF	<i>Disjunctive Normal Form</i>
DOP	<i>Delta-Oriented Programming</i>
DSE	<i>Design Space Exploration</i>
DSL	<i>Domain Specific Language</i>
DTMC	<i>Discrete Time Markov Chains</i>
EA	<i>Evolutionary Algorithms</i>
EDF	<i>Earliest Deadline First</i>

EFSM	<i>Extended Finite State Machine</i>
EM	<i>Extended Module</i>
ES	<i>Embedded Systems</i>
EUF	<i>Equality and uninterpreted functions</i>
FA	<i>Finite Automata</i>
FM	<i>Formal Methods</i>
FMEA	<i>Failure Mode and Effects Analysis</i>
FMECA	<i>Failure Modes, Effects, and Criticality Analysis</i>
FOGD	<i>Fail-Operational Graceful Degradation</i>
FOL	<i>First-order predicate logic</i>
FR	<i>Functional Requirements</i>
FSM	<i>Finite State Machines</i>
FT	<i>Fault-Tolerant</i>
FTA	<i>Fault Tree Analysis</i>
FTD	<i>Fault Tree Diagram</i>
FV	<i>Formal Verification</i>
GA	<i>Genetic Algorithms</i>
GP	<i>Genetic Programming</i>
GPP	<i>Graph Partitioning Problem</i>
gppp	<i>Grammar Guided GP</i>
GRASP	<i>Greedy Randomized Adaptive Search Procedure</i>
HmD	<i>Homogeneous Duplex Pattern</i>
HOOD	<i>Hierarchical Object Oriented Design</i>
HtD	<i>Heterogeneous Duplex Pattern</i>

ILP	<i>Integer Linear Programming</i>
ILS	<i>Iterated Local Search</i>
IMA	<i>Integrated Modular Avionics</i>
KL	<i>Kernighan–Lin</i>
KPN	<i>Kahn Process network</i>
LP	<i>Linear Programming</i>
LS	<i>Local Search</i>
LTL	<i>Linear Temporal Logic</i>
LTS	<i>Labeled Transition System</i>
M-oo-N	<i>M-Out-Of-N</i>
MA	<i>Memetic Algorithm</i>
MA	<i>Monitor-Actuator</i>
MASCOT	<i>Modular Approach to Software Construction, Operation and Test</i>
MBSA	<i>Model-Based Safety Analysis</i>
MC	<i>Markov Chains</i>
MCDM	<i>Multiple Criteria Decision Making</i>
MCS	<i>Minimal Cut-Sets</i>
MGP	<i>Multi-level Graph Partitioning</i>
MILP	<i>Mixed Integer Linear Programming</i>
ML	<i>Machine Learning</i>
MO PB-ILP	<i>Multi-Objective Pseudo-Boolean ILP</i>
MoC	<i>Model of Computation</i>
MOEA	<i>Multi-Objective Evolutionary Algorithms</i>
MOOP	<i>Multi-Objective Optimization Problem</i>

MPSoC	<i>Multi-Processor Systems on a Chip</i>
MTBF	<i>Mean Time Between Failures</i>
MTTF	<i>Mean Time To Failure</i>
MTTR	<i>Mean Time To Repair</i>
NFR	<i>Non-Functional Requirements</i>
NN	<i>Neural Network</i>
NSCP	<i>N-Self Checking Programming</i>
NVP	<i>N-Version Programming</i>
OBDD	<i>Ordered Binary Decision Diagram</i>
OMT	<i>Optimization Modulo Theories</i>
OOP	<i>Object-Oriented Programming</i>
ORM	<i>Object-Role Modeling</i>
OVM	<i>Orthogonal Variability Models</i>
PB-SAT	<i>Pseudo-Boolean Satisfiability</i>
pdf	<i>Probability Density Function</i>
PI	<i>Prime Implicants</i>
PLA	<i>Product-Line Architectures</i>
PLP	<i>Placement Problem</i>
PN	<i>Petri Nets</i>
POS	<i>Product of Sums</i>
PSC	<i>Protected Single Channel</i>
PSO	<i>Particle Swarm Optimization</i>
PTA	<i>Probabilistic Timed Automata</i>
RAMS	<i>Reliability, Availability, Maintainability, and Safety</i>

RAP	<i>Redundancy Allocation Problem</i>
RB	<i>Recovery Block</i>
RBBV	<i>Recovery Block with Backup Voting</i>
RBD	<i>Reliability Block Diagram</i>
RL	<i>Reinforcement Learning</i>
ROBDD	<i>Reduced Ordered Binary Decision Diagram</i>
ROOM	<i>Real Time Object Oriented Modelling</i>
RUP	<i>Rational Unified Process</i>
S-P	<i>Series-Parallel</i>
S2ML	<i>System Structure Modeling Language</i>
SA	<i>Simulated Annealing</i>
SAP	<i>System Assembly Problem</i>
SASDM	<i>Structured Analysis and Structured Design method</i>
SAT	<i>SATisfiability</i>
SC	<i>Sanity Check</i>
SDF	<i>Synchronous dataflow</i>
SE	<i>Safety Executive</i>
SGTS	<i>Stochastic Guarded Transition Systems</i>
SLP	<i>SeLection Problem</i>
SMC	<i>Symbolic Model Checking</i>
SMT	<i>Satisfiability Modulo Theories</i>
SONET	<i>Synchronous Optical NETwork</i>
SOP	<i>Sum of Products</i>
SPL	<i>Software Product Line</i>

SPN	<i>Stochastic Petri Nets</i>
STAMP	<i>System-Theoretic Accident Model and Process</i>
STPA	<i>System-Theoretic Process Analysis</i>
TLE	<i>Top Level Event</i>
TMR	<i>Triple Modular Redundancy</i>
TS	<i>Tabu Search</i>
USDP	<i>Unified Software Development Process</i>
VLSI	<i>Very Large Scale Integration</i>
VML	<i>Variability Modeling Language</i>
WCET	<i>Worst-Case Execution Time</i>
WD	<i>Watchdog</i>
WSP	<i>Workflow Satisfiability Problem</i>
ZBDD	<i>Zero-suppressed Binary Decision Diagram</i>

Chapter 1

Introduction

1.1 Context

Complex systems, like automobiles, airplanes or industrial automation systems, are software-intensive *Embedded Systems* (ES) in which software based sub-systems interact with the physical world using sensors and actuators to fulfill their intended service. Such systems that integrate computation units and physical processes are also called *Cyber-Physical Systems* (CPS) [1]. Day by day, the amount of software in such systems is growing and increasingly interconnected, leading to increasing complexity. Many ES operate in safety-critical environments, like in the automotive domain, avionics domain, or industrial automation domain. Those systems have critical properties in a sense that unhandled malfunctions of system parts may lead to increasing cost or cause damages and unacceptable harms to the system itself, to the physical system environment, and even to the safety of human beings. It is very important therefore that those malfunctions are detected and handled in a safe manner to prevent loss of operation and avoid any form of harm.

1.2 Problem Statement and Motivation

For the above reasons, in today's ES design, reliability and safety are becoming the most important concerns. As a consequence, the configuration of *Fault-Tolerant* (FT) mechanisms is a critical design decision. Indeed, safety critical systems need to contain mechanisms to detect malfunctions of system parts and to react on these malfunctions properly such that no harm can occur with impact to material or life.

In addition, ES may have competing design objectives. Architectures of ES must therefore be configured to meet varied requirements and, in general, multiple design objectives.

Furthermore, a trend is towards mixed-criticality systems [2], meaning that components with different levels of criticality are executed by the same hardware device. It has to be guaranteed that errors of low critical components can never have negative impact on highly critical components. 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).

The motivation of this work is to provide a fully automated approach to the reliability assessment of complex *redundant system architectures* and to formally support the automatic *Design Space Exploration* (DSE) wrt. multiple (conflicting) design objectives, with respect to the given specifications, while ensuring the fulfillment of fail-operational requirements. With a *model-driven* approach [3], complexities of ES are managed at the highest level of abstraction (system level) by using models as key artefacts throughout the development process. We reason at logical level, applying the proposed method before the implementation phase and deployment phase, thus independently from the specific target platform, to be as more general as possible.

1.3 Research Questions and Contribution

In this work, we tackle the following research questions and present contributions to address them.

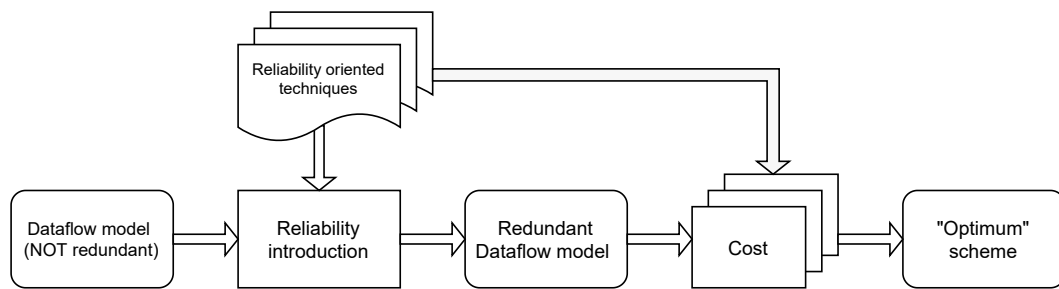


Figure 1.1: Redundant system-level synthesis flow

- RQ1: How to perform the DSE to find an “optimum” redundant schema?
- RQ2: How to formally analyze the ability to keep functional features available in scenarios of failing system elements?
- RQ3: How to introduce redundancy, where needed, at lesser cost? And, more in general, how to perform the assessment of non-functional features at early stages of design?
- RQ4: How to set up a fully automated process?
- RQ5: How to guarantee that the logic system and the physical system are consistent?

Our aim is to propose a methodology for mapping a data-flow based behavioral model onto a target platform whose architecture is formally captured in a high level model, with the goal of optimizing a few system performance criteria. The above contribution can be split in two parts. The first part is the following: given a generic high-level system model, find the best FT resource allocation scheme. Next step is to map the redundant scheme come out from the DSE onto a physical architecture, i.e., a platform dependent system, ensuring consistency between the high-level system model and the actual implementation on the target platform. In this work, we present our contribution to provide a solution to the first part (see Figure 1.1).

1.4 Outline

The residual part of this work is structured as follows. Chapter 2 gives an overview of background notions building the fundamentals of this work. Chapter 3 proposes a critical overview of the related works. The proposed methodology is presented in Chapter 4, Chapter 5, Chapter 6, and Chapter 8. Chapter

7 and Chapter 9 contain experimental evaluation. Chapter 10 summarises conclusions, and suggests future work. Additional material is contained in appendix chapters.

Chapter 2

Background Notions

In this section we present background notions building the fundamentals of this work. The research problem addressed in this thesis crosses several areas where each contains a large body of research. The chapter is organized into distinct sections that provide a brief introduction and an overview of each one of them.

2.1 Reliability Assurance of Complex Systems

The growing complexity of ES, as well as the increasing cost incurred by loss of operation or even worst the damages as a consequence of failures, have brought the aspects of *Reliability, Availability, Maintainability, and Safety* (RAMS) to the forefront. The term reliability appears often for reliability, availability, maintainability, and safety. The expectation is that complex systems are not only free from failures when starting operation, but also perform the required function without failures for a stated time interval, and possibly have a fail-safe behavior, i.e., preserve safety, in case of critical failures. RAMS, and more in general all performance parameters have to be built in during design and development phases, and retained during production and operation of the system.

2.1.1 Basic Definitions

A *system* is an entity that interacts with other entities, i.e., other systems, including hardware, software, humans, and the physical world with its natural phenomena [4]. These other systems constitute the *environment*. The *system boundary* is the frontier between the system and its environment. A *system function* is what the system is intended for. It is described by the *specification* in terms of functionality and performance. The *behavior* of a system is what the system does to implement its function and can be described by a sequence of states. The *service* delivered by a system is its behavior as it is perceived by its users that receive service. A *system failure* occurs when the delivered service deviates from fulfilling the system's function. An *error* is a system state that leads to failure: an error affecting the service is an indication that a failure has occurred. A *fault* is the cause of the error. Faults can be introduced in every phase of the project and they are propagated between phases.

2.1.2 Basic Concepts

This section introduces important concepts used in RAMS engineering and shows their relationships.

- **Reliability:** Experience shows that only a probability of whether a given system will operate without failures during a stated period can be given. This probability is a measure of the system's reliability [5]. Reliability is a characteristic of the system, expressed by the probability that it will perform the required function for a stated time interval. Quantitatively, it specifies the probability that no interruption of the service will occur during that time interval. This does not mean that redundant parts may not fail, as such parts can fail and be repaired on-line (i.e., without operational interruption at system level). The concept of reliability thus applies to both non-repairable and repairable items. An *item* is a functional unit of arbitrary complexity (e.g., component, equipment, subsystem, system) that can be considered as an entity for investigations. It may consist of hardware, software, or both. The *required function* specifies item's task and it is the starting point for any reliability analysis, as it defines failures. The *failure rate* plays an important role in reliability analysis. When a time basis is determined, failures can be expressed in

several ways. The *cumulative failure function* (aka the *mean value function*) denotes the average cumulative failures associated with each point of time. The *failure intensity function* represents the rate of change of cumulative failure function. The *failure rate function* (aka the rate of *occurrences of failures*) is defined as the probability that a failure per unit time occurs in the interval $[t, t + \Delta t]$, given that a failure has not occurred before t . The *Mean Time To Failure* (MTTF) function represents the expected time that the next failure will be observed. It is also known as *Mean Time Between Failures* (MTBF). Another quantity related to time is *Mean Time To Repair* (MTTR), which represents the expected time until a system will be repaired after a failure is observed.

- **Availability:** Availability is a broad term, expressing the ratio of delivered to expected service. It is the probability that a system is available. When the MTTF and MTTR for a system are measured, its availability can be obtained as: $\text{Availability} = \text{MTTF} / (\text{MTTF} + \text{MTTR})$
- **Maintainability:** Maintenance defines the set of actions performed on the item to retain it in or to restore it to a specified state. Maintenance deals thus with preventive maintenance, carried out at predetermined intervals e.g., to reduce wear-out failures, and corrective maintenance, carried out at failure and intended to bring the item to a state in which it can perform the required function. Due to the increasing maintenance cost, maintainability aspects have grown in importance.
- **Safety:** Safety is the ability of the item to cause neither injury to persons, nor unacceptable consequences to material or environment during its use. While *reliability assurance* aims to minimize the number of failures, *safety assurance* examines measures that can bring the item in a safe state at failure (fail-safe procedure). Moreover, for technical safety, effects of external events (human errors, natural catastrophes, attacks, etc.) are important and must be considered carefully. Closely related to the concept of safety are those of risk, risk management, and risk acceptance. Experience shows that risk problems are generally interdisciplinary. The basic steps in proving the safety of a system are shown in Figure 2.1 [6]. The most relevant approaches are presented in the next paragraph.

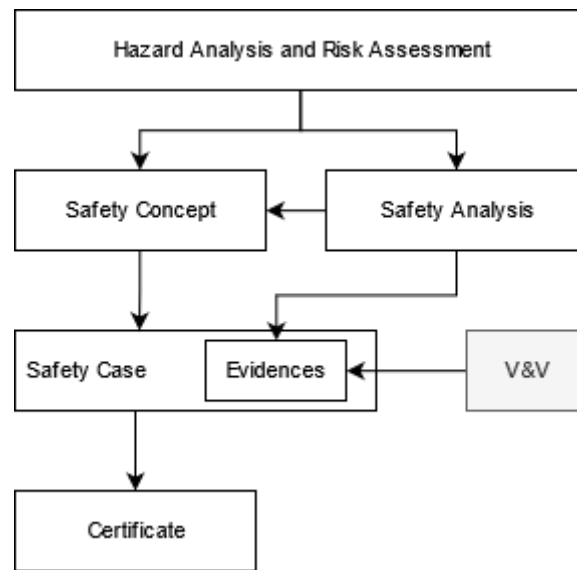


Figure 2.1: Safety engineering lifecycle from Trapp et Al. [6].

2.1.3 RAMS Standards

Customer requirements for reliability can be quantitative or qualitative. Besides quantitative requirements (e.g., MTBF, MTTR, Availability), customers require a quality management system and often also the realization of an assurance program. *Quantitative requirements* are given in system specifications and contracts. They set targets for reliability, maintainability, availability, and safety, as necessary, along with associated specifications for required function and operating conditions. *Qualitative requirements* are covered by national and international standards, and generally deal with quality assurance and management systems. Depending upon the application field (aerospace, nuclear, defense, industrial, automotive), these requirements can be more or less stringent (see Figure 2.2). As stated by Birolini [7], objectives of such standards are, in particular:

- Harmonization of quality assurance and management systems, as well as of terms and definitions.
- Enhancement of customer satisfaction.
- Standardization of configuration, environmental and operating conditions, logistic support, and test procedures, as well as of selection and qualification criteria for components, materials, and production processes.

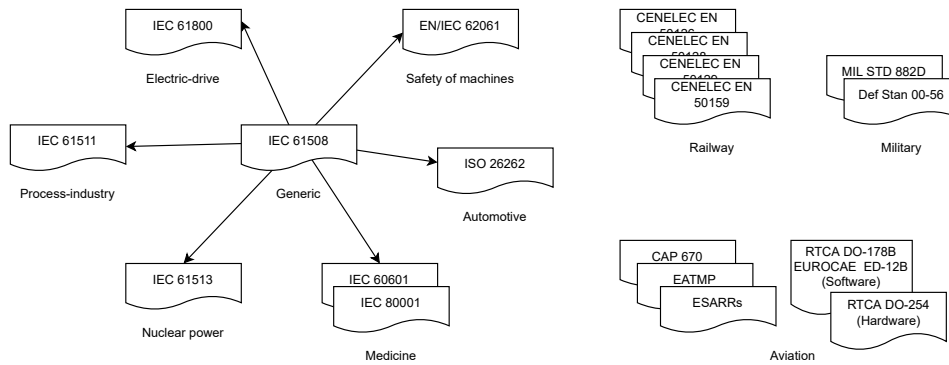


Figure 2.2: Safety regulations, norms, and standards.

2.1.4 Common Functions for Modeling Reliability

The most frequently used function in reliability engineering is the *reliability function* or *probability of success*, denoted by $R(t)$. It represents the probability that a brand new component will survive longer than a specified time. This function gives the probability of an item operating for a certain amount of time without failure. This means that reliability is a function of time. In other words, one must specify a time value with the reliability value. For example, a reliability of 97,5% at 50 hours means that if 1000 new components are put into the field, then 975 of those components are expected to last at least 50 hours of operation. This degree of flexibility makes the reliability function a better reliability specification than the MTTF mentioned above. The *Probability Density Function* (pdf), denoted by $f(t)$ is a continuous representation of a histogram that shows how the number of component failures are distributed in time. The *Cumulative Distribution Function* (CDF) or *unreliability function* or *probability of failure*, denoted by $Q(t)$, represents the probability that a brand new component will fail at or before a specified time. For example, an unreliability of 2,5% at 50 hours means that if 1000 new components are put into the field, then 25 of those components are expected to fail by 50 hours of operation. The *failure rate* function or *instantaneous failure rate* or *hazard rate*, denoted by $\lambda(t)$, represents the probability of failure per unit time t , given that the component has already survived to time t . If any one of the four functions presented above is known, the remaining three can be obtained. Reliability and unreliability functions yield probabilities at a given time from which reliability metrics can be calculated. Plotting the failure rate versus time is an important tool to figure out how a product fails. If the failure rate decreases with time, then the product exhibits infant mortality or early life

failures, typically caused by design errors, poor quality control, or material defects. If the failure rate is constant with time, then the product exhibits a random failure rate behavior. Some possible causes of such failures are higher than anticipated stress, misapplication, or operator error. If the failure rate is increasing with time, then the product wears out. These failures are caused by mechanisms that degrade the strength of the component over time such as mechanical wear or fatigue.

Probability Distributions

To express reliability and model the possible failures quantitatively, probability distributions are needed. One of the most common is the *exponential distribution*. It has only one parameter λ , representing the failure rate of the component or system. Its probability density function is the following:

$$f(t) = \lambda e^{-\lambda t}$$

This leads to the following reliability function:

$$R(t) = e^{-\lambda t}$$

or, dually, to the following unreliability function:

$$U(t) = 1 - e^{-\lambda t}$$

It can be used for the time in which the failure rate is constant. If the failure rate is constant, it means that it is independent of the component's or system's age, i.e., the reliability $R(t)$ represents the probability that the component will not fail within the time interval $(0, t)$. This can be a reasonable assumption for many components that have almost no wear off. This means that for their lifetime in the system, the components have a constant failure rate. One of the main advantages of the exponential distribution is that it simplifies the calculation processes. When the ageing effects have to be included in the model, we can use the *Weibull distribution*. The most general expression of its probability density function is the following three-parameter Weibull distribution expression:

$$f(t) = \frac{\beta(t-\gamma)^{\beta-1}}{\eta^\beta} e^{-((t-\gamma)/\eta)^\beta}$$

where $\beta > 0$ is the shape parameter (aka Weibull slope), $\eta > 0$ is the scale parameter, and γ is the location parameter. Frequently, the location parameter is not used, and the value for this parameter can be set to zero. When this is the case, the pdf equation reduces to that of the so called two-parameter Weibull distribution.

$$f(t) = \frac{\beta t^{\beta-1}}{\eta^\beta} e^{-(t/\eta)^\beta}$$

A value of $\beta < 1$ indicates that the failure rate decreases over time. This happens if there are defective components failing early. A value of $\beta = 1$ indicates that the failure rate is constant over time. A value of $\beta > 1$ indicates that the failure rate increases over time. A change in the scale parameter η has the effect of a change of the abscissa scale on the distribution, i.e., increasing the value of η while holding β constant has the effect of stretching out the pdf.

This leads to the following reliability function:

$$R(t) = e^{-(t/\eta)^\beta}$$

or, dually, to the following unreliability function:

$$U(t) = 1 - e^{-(t/\eta)^\beta}$$

In the fields of dependable computing and safety analysis, one can find fault taxonomies, methods for identifying system faults, methods to analyze their potential impacts, and techniques to remove them. Techniques for dealing with system faults have been proposed, and current state of the art in the field identifies four different ways to increase a system's reliability [8]:

- *Fault avoidance/prevention*: how to avoid/prevent by construction fault occurrence or introduction.
- *Fault detection/removal*: how to detect by verification and validation the presence (number, seriousness) of faults and eliminate them.
- *Fault tolerance*: how to ensure by redundancy a service capable of fulfilling the system's function in the presence of faults.
- *Fault forecasting*: how to estimate by evaluation the presence of faults and the occurrence and consequences of failures.

2.1.5 Failure Classification

The growing complexity of ES, together with the pressure to reduce system development time (aka “time-to-market”), makes the delivery of low-defect systems a challenging and complex activity. Even if the specifications are correctly given and the system successfully verified, there are some problems that can nevertheless lead to a malfunctioning system. For example, it may be the case that the physical components that execute the computations are damaged.

We can organize failure modes in ES into the following main groups [9]:

- **Hardware failures modes**

- Mechanical
 - * Poor quality control during manufacture
 - * Deterioration
 - * Shock
 - * Corrosion
 - * Deformation of components because of temperature
- Electronic
 - * Manufacturing defects
 - * Lack of physical/electrical separation during installation
 - * Interference between subsystems
 - * Issues in the communication between subsystems
 - * Wrong input value from sensors with respect to expected ones
 - * Heat, humidity
 - * Design defects

- **Software failures modes**

- Specification or design defects
 - * Failure to recognize within the specification the full range of circumstances in which the plant must operate
 - * Wrong/inadequate standards used
 - * Inadequate Management of Change (control of plant modifications)
 - * Common ageing processes on redundant channels

- Code rot: accumulated run-time faults.
- Buffer overflow: the computer memory is smaller than the programmer expected, so during operation of the embedded system, one of the programs in the system is accessing wrong parts of the computer's memory.
- Dangling pointers: this error is common in non-safe programming languages in which the human programmer is responsible for making sure that every pointer points to the right memory location at all times.
- Resource leaks in which programming errors lead to the loss of computer control over some of the hardware resources; memory leaks are the simplest form of resource leak.
- Race conditions in which specific relative timing events of different components of the system leads to unexpected behavior. Such race conditions are often hard to detect by testing only.
- Semantic design, for example: the meaning of an arrow between two subsystems in a visual software environment should be the same as the interpretation of it by the hardware.

- **System failures modes**

- Commissioning testing: failure to test adequately all credible circumstances.
- Improper installation
- Maintenance or operations failure
 - * Failure to repair defective equipment in a timely manner.
 - * Maladjustment of set-points, limit switches, etc.
 - * Improper or inadequate maintenance or test procedures.
 - * Failure to follow maintenance procedures.
 - * Poor control of over-rides or interlock defeats.
 - * Poor housekeeping.
 - * Poor quality spare components.
- Environmental aspects

- * Temperature, humidity, vibration, stress, corrosion, contamination (abrasive material, chemical agent, etc.), radio frequency interference, radiation, static charge, extreme weather (rain, snow, hail, ice, wind) seismic event, tsunami.
- Human operators errors
- Intruders: malicious entities (humans and other systems) that attempt to exceed any authority they might have and alter service or halt it.

2.1.6 Failure Modes Analysis

The safety analysis approaches aim to analyze the possible effects of faults in a system. Failure rate analyses presented above basically do not account for the effect (consequence) of a failure. The most relevant approaches to do that are:

- ***Fault Tree Analysis (FTA)***: FTA provides a logical method for graphically presenting the chain of events leading to a system failure, determining system safety and reliability from the event probabilities [10]. It was originally developed in 1961 at Bell Laboratories under an U.S. Air Force Ballistics Systems Division to study the Minuteman Launch Control System [11]. FTA uses Boolean logic to combine a series of events to analyze their effects on a system and understand how a system can fail. It is a deductive top-down method [12] to identify the component level failures (aka *basic events*) that cause the system level failure (aka *Top Level Event* (TLE)) to occur. The failure events are organised into a *Fault Tree Diagram* (FTD). There are numerous FTA symbols, but these are broadly divided in two categories: “events” and “logic gates” that connect the events to identify the cause of the top undesired event (see Figure 2.3).

The set of all fault configurations is also referred to as *Cut-set* (CS). A CS of a fault tree is a combination of basic event occurrences such that the top event occurs. Much of the FTA analysis effort is put on finding all *Minimal Cut-Sets* (MCS). A MCS is a set of basic event occurrences so that the top event occurs, with each of the basic events necessary for that occurrence. For example, the MCS of example FTA in Figure 2.4 is the following:

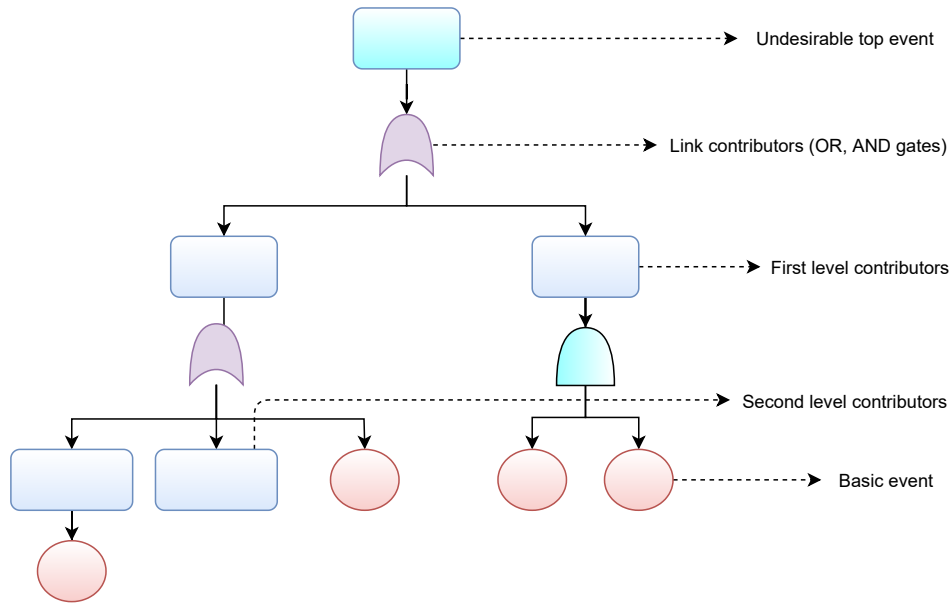


Figure 2.3: FTA symbols

- $MCS_1 = AC = P(A) * P(C)$
- $MCS_1 = AD = P(A) * P(D)$
- $MCS_1 = BC = P(B) * P(C)$
- $MCS_1 = BD = P(B) * P(D)$

TLE occurs if one or more of the minimal cut set occurs:

$$P(TLE) = P(E_1) \cap P(E_2) = P(E_1) * P(E_2)$$

Where:

$$P(E_1) = P(A) \cup P(B) = P(A) + P(B) - P(A) * P(B),$$

$$P(E_2) = P(C) \cup P(D) = P(C) + P(D) - P(C) * P(D)$$

To get the MCS of complex and large fault trees, specific tools implementing algorithms for extraction are needed.

- **Reliability Block Diagram (RBD)**: A RBD is a graph in which system components are connected according to their logical relation of reliability. Each component is represented by a box (aka reliability block) that is assumed to be in operating or failed states. If it is possible to

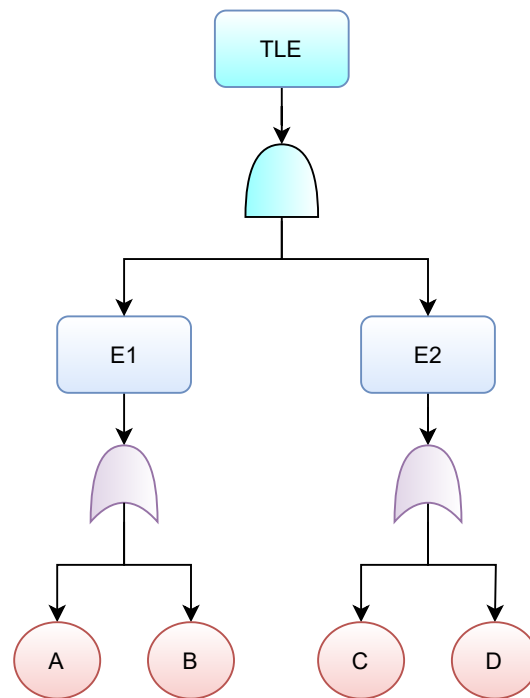


Figure 2.4: Example of fault tree diagram

trace at least one path from the start of the RBD to a specific component through operational components, the specific component is considered operational. It is easy to comprehend because it is simple and has visual impact. It should be noted that a system might require more than one RBD to describe it. As an RBD is a graphical representation of a Boolean expression, and a Boolean expression may take different forms, so similarly there is not a unique RBD for a system. This is particularly true for those systems capable of performing several functions, or experience several different operating states. An RBD may be required for each particular condition. A fault tree may be converted into a RBD and vice versa, as illustrated in Figure 2.5. As for fault trees, traditional solution of RBD involves the determination of the MCS. Figure 2.6 illustrates the RBD equivalent to fault tree of Figure 2.4. The system will fail in the following cases: $\{A, B, C, D \text{ fail}\}$, or $\{A, B, C \text{ fail}\}$, or $\{A, B, D \text{ fail}\}$, or $\{A, C, D \text{ fail}\}$, or $\{B, C, D \text{ fail}\}$, or $\{A, C \text{ fail}\}$, or $\{A, D \text{ fail}\}$, or $\{B, C \text{ fail}\}$, or $\{B, D \text{ fail}\}$. All of these are CSs. However, not all are MCS. For example, the case $\{A, B, C \text{ fail}\}$ is not a minimal CS because, if B is removed, the remaining events are also a CS. This may be more evident by examining the RBD in Figure 2.6.

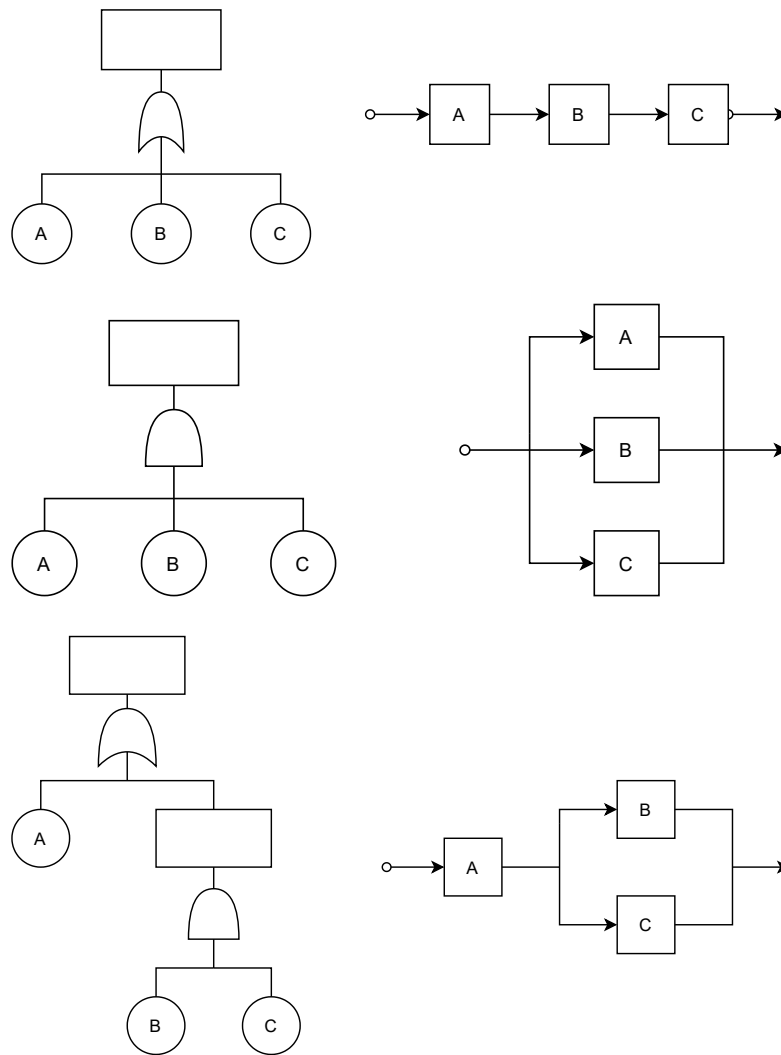


Figure 2.5: Examples of conversion from FTD to RBD

- Failure Mode and Effects Analysis (FMEA):** FMEA is a systematic way of identifying failure modes of a system, and evaluating the effects of the failure modes and criticality on the higher level. It was first developed by the U.S. military in the 1940s, and is now widely used in industries, including aerospace and electronics. A teamwork with designer and reliability engineers performs a bottom-up (inductive) procedure, which is established in international standards [13], [14]. The general benefits of FMEA include prevention planning, cost reductions, ability to identify change requirements, increased throughput, and decreased waste. There are also a few difficulties that need to be taken into consideration. First of all, FMEA only provides assessments, it does not eliminate the problems that it uncovers. In addition, it relies on team

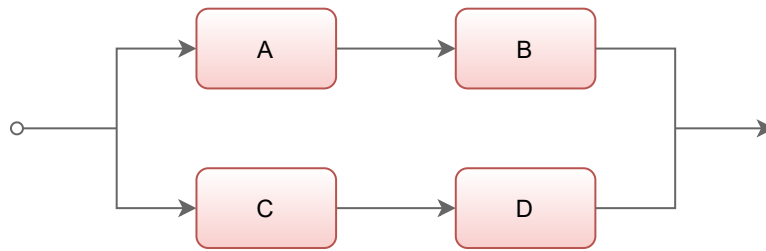


Figure 2.6: RBD equivalent to fault tree of Figure 2.4

experience: the more experienced the team, the better the FMEA will be. Moreover, effective FMEA requires each step of a process to be examined.

- ***Failure Modes, Effects, and Criticality Analysis (FMECA):*** FMECA builds upon the FMEA process, and not only it identifies potential failure modes, but also investigates and isolates any potential failure through a series of actions. FMECA has the advantage of being more comprehensive than FMEA by establishing relationships between failure causes and effects, and the criticality of corrective actions. As with FMEA, FMECA comes with its own difficulties that need to be taken into account. These include the amount of labour required, and the difficulty in assessing multiple-failure or cross-system effects. FMECA also does not typically consider software or human interaction implications. Both FMEA and FMECA can be used to help fulfil quality and safety requirements.
- ***System-Theoretic Process Analysis (STPA):*** STPA is a modern hazard and safety analysis technique, based on the accident causation model (*System-Theoretic Accident Model and Process (STAMP)*, [15]), applicable early in the design process of a system to achieve an acceptable risk level [16]. Since accidents are more than a chain of events, but they rather involve complex dynamic processes, STAMP treats accidents as a control problem (not a failure problem) and tries to prevent accidents by enforcing constraints on component behavior and interactions. It captures more causes of accidents, such as component failure accidents, unsafe interactions among components, human behavior, software behavior, design errors, and flawed requirements. The aim is to identify the potential hazardous causes in complex safety-critical systems at different architecture abstraction levels [17]. To this aim, i.e., to find inadequate

control in a design, the STPA approach combines safety analysis and software test case generation.

2.1.7 Concept of Redundancy

High reliability at system level can often only be reached with the help of redundancy. Redundancy is the basic idea of FT, which is intended to preserve the delivery of correct service in the presence of active faults [4]. *Redundancy* is the existence of more than one means for performing the required function. If a part of the system can suffer a fault, there should be a redundant part of it that can cover for the fault and thus avoid a failure. The components of the ES can be placed in either active or standby mode.

In the *active configuration*, all of the redundant components operate simultaneously as soon as the system's mission starts.

Standby configuration can be considered in three variants: cold, warm, and hot standby. In the *cold standby* configuration, if the primary component is operational and not experiencing any issues, the redundant ones are not powered and not operating. In case of a failure in the primary component, the redundant ones begin the starting procedure, and when it terminates the components are exchanged. In the *warm standby* configuration, the redundant components are powered, but not operating. The primary and redundant components are synchronized when a failure occurs in the main one. When this operation terminates they are switched. A warm standby approach guarantees higher availability than a cold standby because the starting procedure has already been performed. In the *hot standby* configuration, the redundant components are powered and ready to be switched seamless into service upon detection of a failure in primary component. The switching mechanism from main to redundant components is much faster than both cold and warm standby approaches, but redundant components can fail autonomously and with the same failure of primary ones. As a result, in the cold standby configuration the system remains reliable while in standby because the redundant components do not fail before they are put into operation. In warm and hot configuration, the system can fail also from its standby state.

2.1.8 Design Patterns for Reliability

The concept of design patterns is a universal approach to describe common solutions to widely recurring design problems. A design pattern is an abstract representation for how to solve a general design problem that occurs over and over in many applications. We can organize most common design patterns as follows, with respective examples. For further information see also the works of de Matos et al. [18], and Armoush [19].

Hardware Patterns

- ***Homogeneous Duplex Pattern (HmD)***: Protection against random faults (aka Homogeneous Redundancy, Standby-Spare Pattern, Dynamic Redundancy, Two-Channel Redundancy).
- ***Heterogeneous Duplex Pattern (HtD)***: Protection against random and systematic faults without a fail-safe state (aka Heterogeneous Redundancy Pattern, Diverse Redundancy Pattern).
- **TMR**: Protection against random fault with continuation of functionality (aka 2-oo-3 Redundancy Pattern, Homogeneous Triplex Pattern).
- **M-oo-N**: Protection against random fault with continuation of functionality.
- ***Monitor-Actuator (MA)***: Protection against random and systematic faults with a fail-safe state.
- ***Sanity Check (SC)***: Lightweight protection against random and systematic faults with a fail-safe state.
- ***Watchdog (WD)***: Very lightweight protections and timebase fault and detection of deadlock with a fail-safe state.
- ***Safety Executive (SE)***: Safety for complex systems with complex mechanisms to achieve failsafe states.

Software Patterns

- ***Voting*** Techniques (Majority Voting, Plurality Voting, Consensus Voting, Maximum Likelihood Voting, Adaptive Voting).

- ***N-Version Programming (NVP)***: software diversity.
- ***Recovery Block (RB)***: fault detection with acceptance tests and backward error recovery.
- ***Acceptance Voting (AV)***: NVP + acceptance test used by the RB.
- ***N-Self Checking Programming (NSCP)***: NVP + self-checking.
- ***Recovery Block with Backup Voting (RBBV)***: RB + NVP.

Hardware and Software Patterns

- ***Protected Single Channel (PSC)***: Safety without heavyweight redundancy, protection against transient faults through checks and monitoring at different points.
- ***3-Level Safety Monitoring (3-LSM)***: MA + WD. Actuation, monitoring and control element.

2.1.9 Dealing with System Faults

In the fields of dependable computing and safety analysis, one can find fault taxonomies, methods for identifying system faults, methods to analyze their potential impacts, and techniques to remove them. Techniques for dealing with system faults have been proposed, and current state of the art in the field identifies four different ways to increase a system's reliability [8]:

- ***Fault avoidance/prevention***: how to avoid/prevent by construction fault occurrence or introduction.
- ***Fault detection/removal***: how to detect by verification and validation the presence (number, seriousness) of faults and eliminate them.
- ***Fault tolerance***: how to ensure by redundancy a service capable of fulfilling the system's function in the presence of faults.
- ***Fault forecasting***: how to estimate by evaluation the presence of faults and the occurrence and consequences of failures.

2.1.10 Dealing with Non-Functional Requirements

The system development process begins with the definition of the high-level system goals based on stakeholder inputs. These high-level goals lead the development team to base a system on a certain architecture style. The specification should be carefully written so that it reflects the customer requirements, and understandable enough so that someone can verify that it meets system requirements. The specification says what things the system does, but it does not say how. Describing how the system implements those functions is the purpose of the architecture. The above concepts only make sense with respect to a given set of requirements. A key point is the separation of requirements into functional and non-functional, where the former describe what a system does (in terms of the relation between inputs and outputs), while the latter consists of properties related to how a system operates, including efficiency, quality of service, and maintenance issues. One specific class of increasingly important functional requirements concerns the security of systems. As stated by some authors [20], we must remark that security requirements have contributed to boosting the importance of formal methods (see Section 2.3). A second specific class of requirements concerns safety issues. *Functional Requirements* (FR) are those services that the system is expected to provide to actors in its environment. *Non-Functional Requirements* (NFR) are those indices that evaluate the quality of the design in terms of reliability, safety, cost, speed, size, etc. FR are usually being taken under consideration at the early stage of process development (architectural level), while NFR are being focused at the end of the project, which does not fulfill the desired qualities [21]. Early design decision is very important to achieve a strong connection between design and requirements, quality of a system, and a consistent product. Hence, FR and NFR should be treated together, at early stages of design.

2.2 Architecture-based reliability evaluation

Since early design decision is very important to achieve quality of a system, the architecture leverages a crucial role. Reliability is one of the essential quality requirements of software systems, especially for life critical ones. A failure to identify certain concerns such as reliability earlier in the development process can result in complete failure of the project. Testing activity is often postponed until too late in the development process, especially on projects

using traditional development life cycles. This translates into increasing cost incurred by loss of operation or even worse damages as a consequence of failures. For this reason, numerous techniques have been developed to evaluate the reliability of a component-based system from the architecture, before the actual system is built. Both qualitative and quantitative methods have been developed to evaluate the quality of a system from its architecture. Qualitative methods include non-numerical approaches such as peer reviews, and practices that usually rely on domain experts and on a substantial amount of manual activity. Quantitative methods include simulations, mathematical evaluations, and formal analysis techniques that aim to get metrics on the quality of the architecture.

2.2.1 Combinatorial Models

Using a combinatorial approach to model reliability translates in decomposing a complex system into functional subsystems, in order to understand how the relationships among them affect system operation. These formalisms create a model equivalent to a Boolean expression.

Reliability Block Diagrams

This method can be used in both design and operational phase to identify poor reliability and provide targeted improvements. It enables the analysis of the effect of component failures on various system configurations, as shown in Figure 2.7. The simplest form of a system for reliability analysis is one where the elements are connected in series. In this case the reliability of the system is the probability that all components succeed.

$$R_S = \prod_{i=1}^N R_i$$

In parallel configuration, at least one component must succeed in order the system succeeds. The probability of failure of a system with parallel components is given by the probability that all components will be simultaneously in the failure state.

$$R_S = 1 - \prod_{i=1}^N (1 - R_i)$$

The probability of failure of a system with a k-out-of-n parallel configuration is the following:

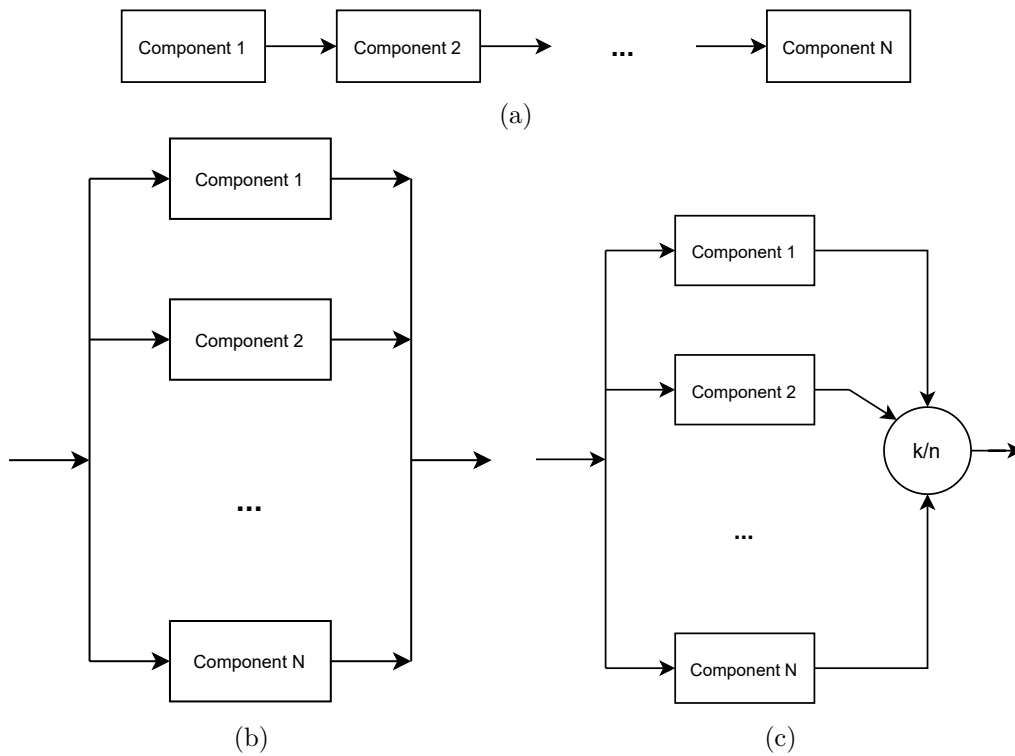


Figure 2.7: Reliability block diagrams for series (a), parallel (b), and k-out-of-n (c) systems.

$$R_s = \sum_{i=k}^n \binom{n}{i} R^i (1 - R)^{n-i}$$

In many cases, it is not easy to recognize which components are in series and which are in parallel. This is what we call a *complex system*, i.e., a system that cannot be broken down to groups of series and parallel components. This complicates the problem of determining the system's reliability. Every path from a starting point to an ending point should be considered. As long as at least one path from the beginning to the end of the path is available, the system has not failed. Hence, the reliability of the system is the probability of the union of these paths.

For example, an inspection of the reliability-wise configuration of system in Figure 2.8 reveals that any of the following failures will cause the system to fail:

- Failure of components A and B
- Failure of components C, D, and E
- Failure of components F and G

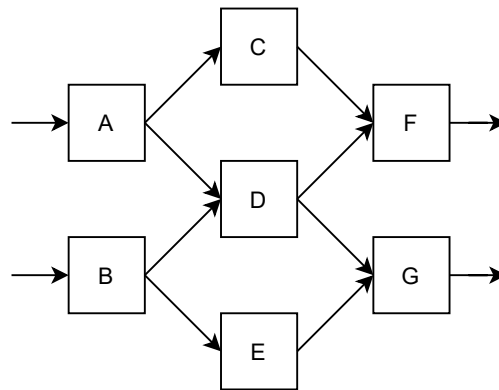


Figure 2.8: Complex system composed by seven components

- Failure of components A, C, and F
- Failure of components A, D, and F
- Failure of components A, D, and G
- Failure of components B, D, and F
- Failure of components B, D, and G
- Failure of components B, E, and G

Leading to the configuration in Figure 2.9.

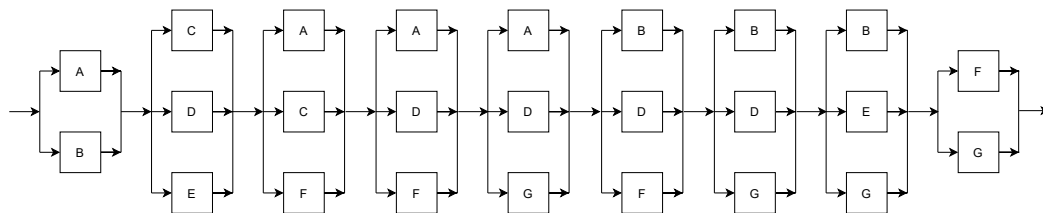


Figure 2.9: Reliability block diagram for system in Figure 2.8

Fault Tree Analysis

As stated above, FTA provides an alternative methodology to RBD for reliability and safety analysis. Failure events are organised into a tree structure. Figure 2.10 illustrates the fault trees equivalent to the RBDs in Figure 2.7. Figure 2.11 illustrates the fault tree for system in Figure 2.8. The main difference between RBD and FTD is that the RBD looks at success combinations whereas the FTD looks at failure combinations. In addition, fault trees have

traditionally been used to analyze events that have a fixed probability of occurring, while RBDs may include time-varying distributions for success or failure of a block. However, a fault tree can be easily converted to an RBD, while it is generally more difficult to convert an RBD into a fault tree, especially in case of very complex configurations.

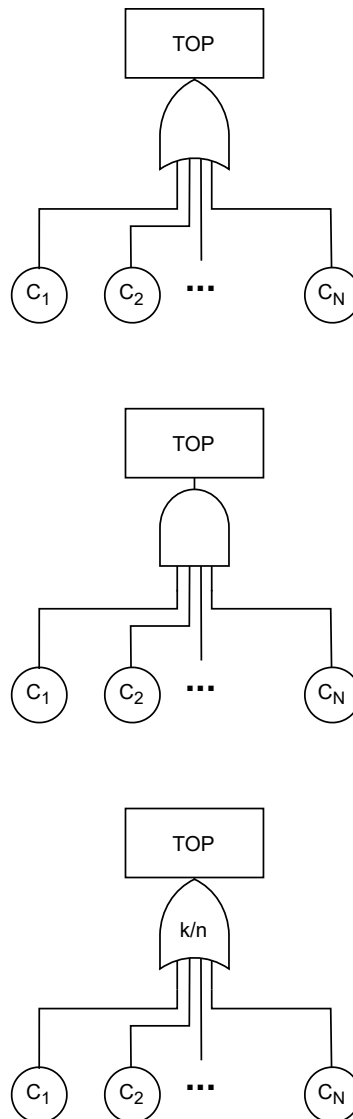


Figure 2.10: Fault trees for series (a), parallel (b), and k-out-of-n (c) systems.

Binary Decision Diagram

A BDD is a data structure for representing Boolean functions. A Boolean function can be represented as a rooted, directed, cyclic graph, which consists

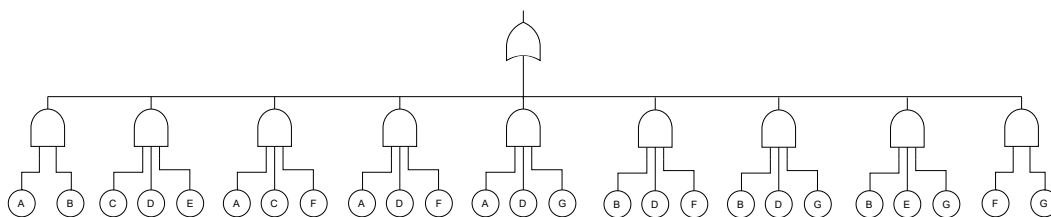


Figure 2.11: Fault tree for system in Figure 2.8

of several decision nodes and terminal nodes. There are two types of terminal nodes called 0-terminal and 1-terminal. Each decision node N is labelled by boolean variable V_N and has two child nodes called low child and high child. The edge from node V_N to a low or high child represents an assignment of V_N to 0 or 1. If the order of the variables we test is fixed, we refer to BDD as *ordered*. In addition, it is *reduced* if the following two properties hold:

- Irredundancy: the low and high successors of every node are distinct.
- Uniqueness: there are no two distinct nodes testing the same variable with the same successors.

A BDD ordered is referred to as OBDD. A BDD ordered and reduced is referred to as *Reduced Ordered Binary Decision Diagram (ROBDD)*. Often BDD representations are exponentially more concise than Boolean formulae in *Conjunctive Normal Form (CNF)*, where CNF is a conjunction of one or more clauses, and a clause is a disjunction of literals. This enhances the effectiveness of memorization techniques. In general, the chosen variable ordering makes a significant difference to the size of the ROBDD representing a given function. Bryant [22] observed that ROBDDs are a canonical representation of boolean functions. This means that for a fixed variable ordering, each boolean function has a canonical (i.e., unique) representation as an ROBDD. We could therefore test the equivalence of two boolean functions by comparing their ROBDDs and checking if they are equal. For example, let F be a boolean formula that depends on the variable x . According to the Shannon decomposition, there exists two formulae, F_0 and F_1 , that do not depend x such that:

$$F = v \wedge F_0 + \neg v \wedge F_1$$

By choosing a total order over the variables, and applying recursively the Shannon decomposition, the truth table of the formula can be graphically represented as a binary tree. The Shannon tree for the formula $F = a \wedge b + \neg a \wedge c$

and the lexicographic order is illustrated in Figure 2.12. The *Binary Decision Tree* (BDT) can be transformed into a BDD by reducing it according to the two following reduction rules (see Figure 2.13):

- **Elimination:** a node with two equal successors is useless and can be eliminated, it is equivalent to its unique successor: $v \wedge F + \neg v \wedge F = F$.
- **Isomorphism:** since two isomorphic subtrees encode the same formula, one is useless, hence, we can merge isomorphic subtrees.

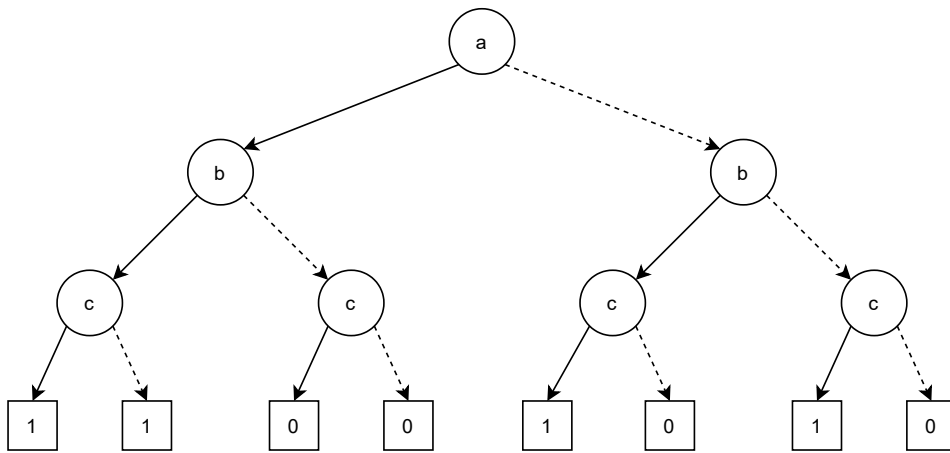


Figure 2.12: Binary tree of the example formula $F = a \wedge b + \neg a \wedge c$

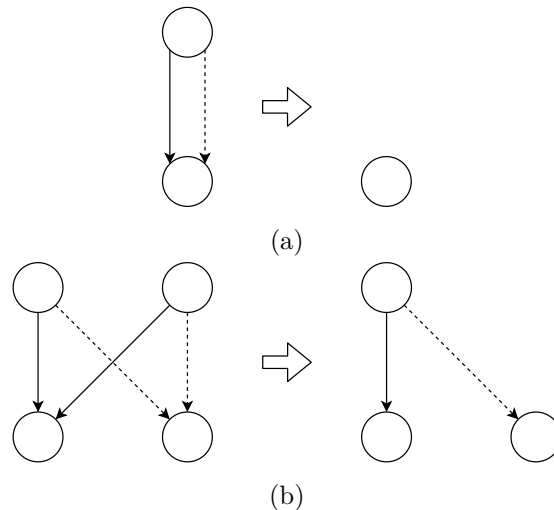


Figure 2.13: Binary tree reduction rules: (a) Elimination, (b) Isomorphism.

The resulting BDD is illustrated in Figure 2.14.

Figures 2.15, 2.16, and 2.17 illustrate the BDDs equivalent to the RBDs in Figure 2.7. The series system works iff all system components are working and, therefore, only one path in the BDD representing this structure function has to end in the 1-labeled sink node, and this path has to contain all n state variables and leaves each of them via 1-labeled edge. A parallel system is functioning iff at least one of the system components is working. It has to contain therefore only one path that ends in the 0-labeled sink node, and this path has to go through all n state variables and leaves each of them via 0-labeled edge. To model the BDD for a binary k -out-of- n , we have to remind that it is functioning if at least k components are working. This implies that, for a given state, at least k components have to be checked if we want to decide whether the system is working or not. This means that every path ending in the sink node 1 has to contain at least k non-sink nodes from which exactly k nodes have to be left via 1-labeled edges. If one or more checked components are in state 0, then we have to investigate other components. Obviously, if we check $n-k+1$ components, and all of them are in state 0, then the system has to be in state 0; in other words, every path containing $n-k+1$ 0-labeled edges has to be terminated by 0-labeled sink node. Figure 2.17 depicts a k -o-o- n system, for $k = 3$ and $n = 5$.

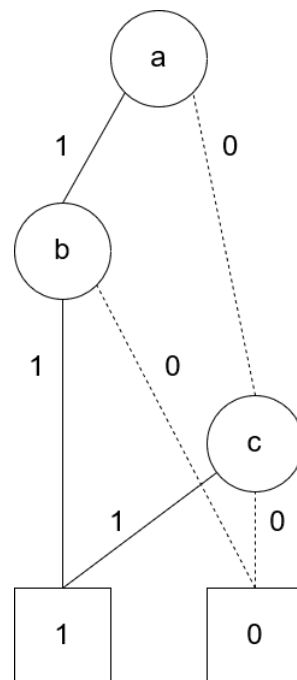


Figure 2.14: BDD of the example formula $F = a \wedge b + \neg a \wedge c$

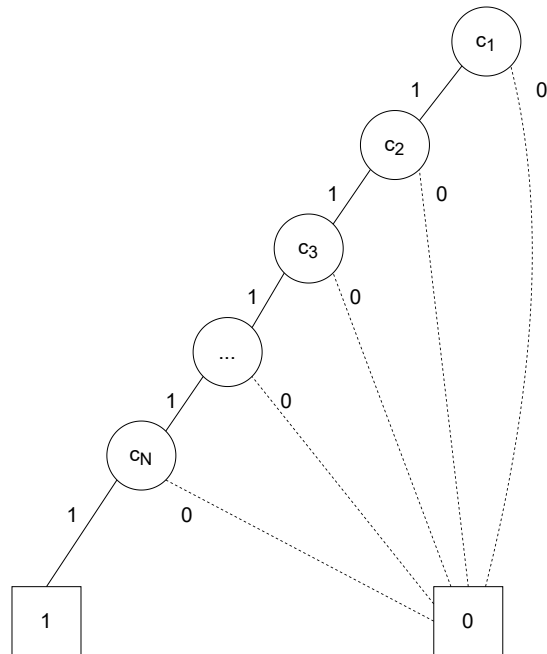


Figure 2.15: Example of a series system in the form of a BDD

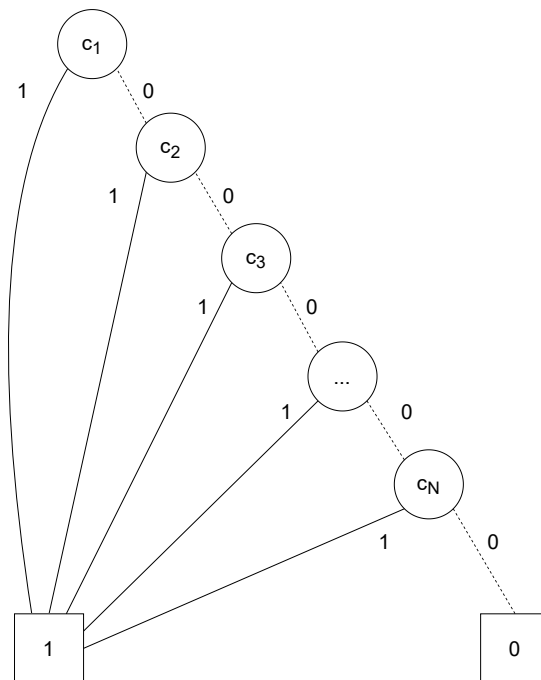


Figure 2.16: Example of a parallel system in the form of a BDD

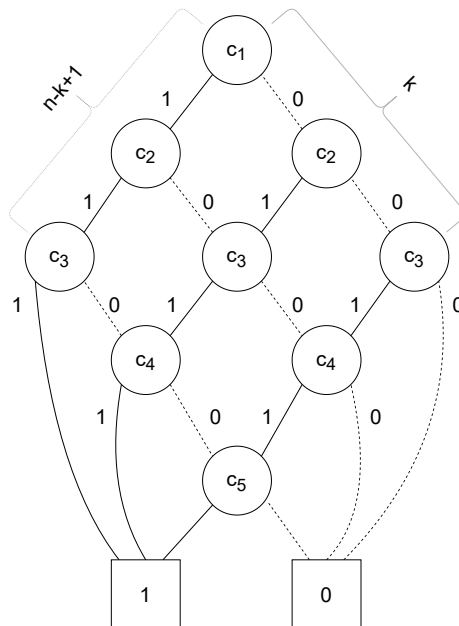


Figure 2.17: Example of a k-o-o-n system in the form of a BDD, with $k=3$ and $n=5$.

FTA is usually performed in two steps: firstly, the MCS of the model are determined by some top-down or bottom-up algorithm, then, some probabilistic quantities of interest are assessed from MCS. With BDD, the methodology is different: firstly, the BDD of the model is constructed. Then, another BDD, often a *Zero-suppressed Binary Decision Diagram* (ZBDD) [23], is built from the first one to encode MCS. This approach has a drawback: the construction of the BDD must be feasible and, as many Boolean problems, it is of exponential worst case complexity. Indeed, the size of the data structure may vary between a linear to an exponential range depending on the variable ordering [22]. Top event probabilities can be assessed either from BDD or from ZBDD. Since these data structures are based on different decomposition principles, algorithms used in each case are different. Exact computations are usually performed with BDD.

2.2.2 State-based Models

An alternative to combinatorial formalisms are state-based formalisms. They can model different states for components and the global system. They can also include effects like failure propagation by adding fitting transitions in

the chain. Effects like delayed repair or different standby strategies can be considered as well.

Markov Models

A common approach to reliability analysis of complex systems uses the *Markov Chains* (MC) model. This analysis is suitable for those systems whose components can be in two states, failed or not failed, and transitions from one state to another can happen from time to time. The analysis can be applied in cases where the change of state at a certain instant does not depend on previous events (i.e., for memory-less system) and the probabilities are known for the transition from operable state to failed state and vice versa. A Markov model is uniquely determined by a set of equations that describes the probabilistic transitions among the states and initialisation probability distributions of the starting states. We have to define each possible state that the system can be in at any given time, and also the transition probabilities per step that link the states together. Mathematically, we can represent the initial state probabilities as a vector $x(\vec{0})$ such that x_i represents the initial probability of being in state i :

$$x(\vec{0}) = (x_1 x_2 \dots x_i).$$

The transitions between the states can be represented by a transition matrix:

$$\vec{T} = \begin{pmatrix} P_{11} & P_{12} & \dots & P_{1i} \\ P_{21} & P_{22} & \dots & P_{2i} \\ \dots & \dots & \dots & \dots \\ P_{i1} & P_{i2} & \dots & P_{ii} \end{pmatrix}$$

where, for example, the term P_{12} is the transition probability from state 1 to state 2. If we want to know the probability of being in a particular state after n steps, we can use the Chapman-Kolmogorov equation to arrive at the following equation:

$$x(\vec{n}) = x(\vec{0})\vec{T}^n.$$

The structural representation of a MC is a directed graph (while the combinatorial models presented in Section 2.2.1 are based on tree structures). When using probabilities and steps the MC is referred to as *Discrete Time Markov Chains* (DTMC), while a MC that uses rate and the time domain is referred

to as a *Continuous Time Markov Chains* (CTMC). Both variants of Markov models that are used in architecture-based quantitative evaluation of reliability. As a simple example, consider a static TMR system without repair. Assuming the ideal voter, and the same failure rate λ for its components, its reliability Markov model is depicted in Figure 2.18. State “3” represents the initial condition of three working components. State “2” indicates the condition of one failed component. State “F” represents the failure state because a majority of the modules in the system have failed.

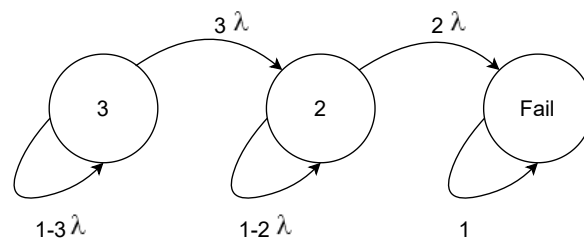


Figure 2.18: General Markov model of a TMR system

MC is much more powerful than RBD and FTA, but it has also its disadvantages: MC cannot be created hierarchically. With FTA or RBD, the model of the TMR system is easily modifiable, for instance it is easy work to replace a component by a more complex system or to add another component. MC cannot be changed that easy as the state space is altered dramatically by such a step. In addition, the size of the MC grows exponential with the number of components in the basic system. Furthermore, MC also lacks intuitivity: it is almost impossible to recognize the basic system architecture of a system by looking at the model.

Petri Net Models

RBD and FTA are both static tools, they cannot capture the state-dependent behavior of system failure mechanisms. Although MC can cope with state-dependent behaviors, it has to face with the state space explosion problem when the system is large and complex. Furthermore, the solution of the differential equations for the MC is a cumbersome work. An alternative tool that has been widely applied in the last decades for discrete event system simulation is *Petri Nets* (PN). The PN is a mathematical model of a parallel system, in the same way that the finite automaton is a mathematical model of a sequential system. A PN is a directed bipartite graph that has two types

of elements, places and transitions, depicted as white circles and rectangles, respectively. A place can contain any number of tokens, depicted as black dots. A transition is enabled (i.e., can be fired) if all places connected to it as inputs contain at least one token. This modeling language can be used to model dynamic systems, and permits the reliability analysis of complex systems. In particular, it permits modeling a system when one or more of the elements are in a degraded state or under repair, as it is possible to include elements of the system that are neither function or failed. Petri net modeling is therefore useful when the repair times are long compared to operating times, as RBDs and FTA assume short or negligible repair times, in most cases. Specifically, widely-used state based formalism are *Stochastic Petri Nets* (SPN), which are a form of PN where the transitions fire after a probabilistic delay determined by a random variable. Figure 2.19 shows a SPN modeling the TMR-system (with repair). PN are easier to handle than MC, as they support modularized modeling to a certain degree. Anyway they have some limits: new components without dependencies can be added easily, while adding dependencies into an existing model can lead to a total different structure of the PN.

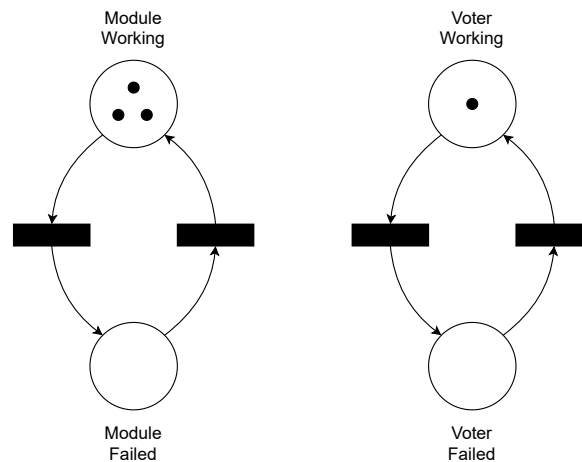


Figure 2.19: Petri Net model of a TMR system

To sum up, state-based formalisms allow us to model effects like failure propagation, standby strategies, and repair. However, they lack the immediacy of combinatorial approaches. While being more powerful, MC and PN cannot compete with the high level approaches of combinatorial approaches regarding the intuitiveness and understandability. Creating, understanding, and mastering models based on state-based methods is more difficult than using combinatorial formalisms, and especially for large models this can be a

problem. Furthermore, efficient algorithms exist for evaluating combinatorial formalisms. The main reason for this is that these formalisms can be translated into BDDs, which are very efficient and exponentially more concise than Boolean formulae. BDDs allow us to perform a qualitative and quantitative analysis in order to determine which component failures will lead to a system failure by analyzing the MCS.

2.2.3 Simulation-based Approaches

For a considerable number of parameters, the platform dependent system evaluations are based on estimations that use field data retrieved during operational usage, historical data from similar systems, or reasonable guesses done by experts. In practice, parameters are hard to estimate accurately. Analytical methods often assume the parameter distributions are normal, and variations can be characterised by the mean and variance alone. More in general, the actual distributions are hard to determine. Hence, a further approach for architecture-based reliability evaluation uses simulation techniques, which investigate the impact of specific components on system reliability and provide system-specific performance and attributes.

2.3 Formal Methods, Techniques, and Tools

Formal Methods (FM) are mathematical techniques, often supported by tools, for developing software and hardware systems. Mathematical rigor enables users to analyze and verify these models at any part of the program life-cycle: requirements engineering, specification, architecture, design, implementation, testing, maintenance, and evolution. FM are used in *specifying* software: developing a precise statement of what the software is to do, while avoiding constraints on how it is to be achieved [24]. Complex software systems require careful organization of the architectural structure of their components: a model of the system that suppresses implementation detail, allowing the architect to concentrate on the analyses and decisions that are most crucial to structuring the system to satisfy its requirements [25], [26]. FM are also used in software *design*. For example, see the early paper by Hoare [27], methods like VDM [28], and in program refinement calculi [29], [30], [31], [32]. At the *implementation* level, FM are used for code verification. The inductive assertion method of program verification was invented by Floyd and Hoare [33], [34],

and involves annotating the program with mathematical assertions, which are relations that hold between the program variables and the initial input values, each time control reaches a particular point in the program. Code can also be generated automatically from FM. FM are also used in software *maintenance* [35] and *evolution* [36].

2.3.1 Computational Models

With the development process, a key point is the choice of a specific method and an appropriate language to represent system under study along all its life cycle. Different methods and languages have been developed for the analysis and design of ES, each with a particular focus on some aspects of the system. Each designer will therefore have to choose the method or language to use depending on the specificities of the application domain, and the level of detail he deems important for his model. In order to manage the complexity and heterogeneity of modern ES, the design approach should be based on the use of one or more FM to describe the behavior of the system at a high level of abstraction, before a decision on its decomposition into hardware and software is taken. The mapping and implementation of the specification must be more automatic, faster, and less error-prone as possible. *Abstraction* is a key attribute of modeling: by removing unnecessary and unwanted details, the complexity of the modeled object is reduced, allowing to effectively engineer systems by successively building more detailed models of the system under development. First of all, in the following the difference between methods and languages is reported.

Methods

A *method* is a process whose objective is to allow to formalize the stages of development of a system, to make this development more faithful to the needs of the client. A language offers a notation, but do not have a process. In a method, one uses one or more modeling languages for symbolic or schematic representation of the various models produced to drive the development and tools that support it. In Table 2.1, we have listed design methods for real-time ES [37], [38], [39], [40], [41]. There is no single method that is overall the best method. Although a large number of successful frameworks have been developed during the last several years, designing a high-quality framework is still a difficult and complex task.

Table 2.1: Examples of design methods

Method	Description	Easy of use	Notation	Code generation	Free tools	System design	Component model	Hierarchies	PRO	CON	Last known activity
<i>Design Approach for Real Time Systems (DARTS)</i>	Highly structured modular system with well-defined interfaces and reduced coupling between tasks	NO	graphic textual	NO	NO	NO	YES	YES	Structured analysis of data	Difficult to control the time	2007
<i>Hierarchical Object Oriented Design (HOOD)</i>	It includes object types which enable common hard real-time abstractions to be represented	YES	graphic textual	YES	NO	NO	YES	YES	Reusability	Complex to use	2016
<i>Modular Approach to Software Construction, Operation and Test (MASCOT)</i>	Developed under the auspices of the United Kingdom Ministry of Defence, it emphasises the architectural aspects of a project rather than functionalities	NO	graphic textual	YES	NO	NO	NO	YES	High level of abstraction	Requirements analysis not directly supported	2016
<i>Object-Role Modeling (ORM) aka NIAM</i>	Used principally for data modeling	NO	graphic	NO	YES	NO	YES	YES	It facilitates implementing the Domain Model pattern	It is often difficult to grasp the shape or purpose of a particular drawing	2016
<i>Real Time Object Oriented Modeling (ROOM)</i>	The initial focus was on telecommunications, even though ROOM can be applied to any event-driven real-time system	NO	graphic textual	YES	YES	NO	NO	YES	Behaviour specification	Requirements phase	2016
<i>Structured Analysis and Structured Design method (SASDM)</i>	It is a waterfall method for the analysis and design of information systems and contrasts with more contemporary agile methods such as DSDM or Scrum	YES	textual	NO	N.A.	YES	NO	NO	It does not rely on a single technique	It does not cover deployment, implementation and testing	2016
<i>Unified Software Development Process (USDF)</i>	It is a popular iterative and incremental software development process framework. The best-known and extensively documented refinement of the Unified Process is the <i>Rational Unified Process</i> (RUP)	YES	graphic textual	NO	YES	YES	NO	YES	Checkpoints to check the quality of artifacts to be delivered in each phase	Consistency problems (semantic multiple interpretations)	2019

Modeling Languages

A language is a set of symbols, rules for combining them (its syntax), and rules for interpreting combinations of symbols (its semantics). A design (at all levels of the abstraction hierarchy from functional specification to final implementation) is generally represented as a set of components, which can be considered as individual blocks, interacting with each other and with an environment that is not part of the design. The *Model of Computation* (MoC) defines the behavior and interaction of these blocks. Thus, a MoC governs the interaction of components in a design. In Table 2.2, we classify and analyze several models of computation that have been used to describe ES with precise mathematical meaning, so that both the validation and the mapping from the initial description to the various intermediate steps can be carried out with tools of guaranteed performance.

Architectural Modeling

According to the ISO/IEC/IEEE 42010:2011 Systems and software engineering - Architecture description standard [42] [42], an *Architecture Description Language* (ADL) is any form of expression for use in architecture descriptions. Thus, we can define ADLs as computer languages describing the software and hardware architecture of a system. Compared to modeling languages that are more concerned with the behaviors of the whole rather than of the parts, ADLs concentrate more on representation of components. An ADL can be narrowly focused, defining a single model kind to frame some concerns, or widely focused to provide several model kinds, optionally organized into viewpoints. One advantage of ADLs is that they represent architectures in a formal way. In addition, they are supported by automated tools for the management of the models. Furthermore, they are designed to be readable to both humans and machines. A disadvantage is that there is not an agreement of what the ADLs shall represent, especially when it comes to the behavior of the system. A set of common ADLs is reported and analyzed in Table 2.3 [43], [44], [45], [46], [47], [48].

Table 2.2: Common types of formal models, usually referred to as model of computations.

Model	Examples	PROs	CONs
State-Based	<ul style="list-style-type: none"> - <i>Finite State Machines</i> (FSM) - <i>Finite Automata</i> (FA) - <i>Extended Finite State Machine</i> (EFSM) 	<ul style="list-style-type: none"> - Hierarchy allows nesting of AND-, OR-, superstates - Large number of commercial tools available - Available back-ends translate model into code 	<ul style="list-style-type: none"> - Generated programs inefficient or difficult to read - Not useful for distributed applications - No program constructs, no object-orientation - No description of non-functional behavior
DataFlow	<ul style="list-style-type: none"> - <i>Kahn Process network</i> (KPN) - Task graphs - <i>Synchronous dataflow</i> (SDF) 	<ul style="list-style-type: none"> - Very natural way of describing real life applications - More efficient use of concurrent resources because becomes easy to parallelize - The scheduling algorithm does not affect the functional behavior 	<ul style="list-style-type: none"> - Difficult to schedule because of need to balance relative process rates - System inherently gives the scheduler few hints about appropriate rates - Expensive and fussy to implement
Process Algebra	<ul style="list-style-type: none"> - <i>Communicating Sequential Processes</i> (CSP) - <i>Communicating Shared Resources</i> (CSR) 	<ul style="list-style-type: none"> - Concurrency - Security 	<ul style="list-style-type: none"> - Communication: latency (network distance, message size), contention (network bandwidth), overhead (interfaces and protocols used) - Synchronization is another source of overhead - Load balancing (non-uniform tasks, starvation)
Logic-Based	<ul style="list-style-type: none"> - Temporal logic - CSR 	<ul style="list-style-type: none"> - It involves logical propositions whose truth values depend on time 	<ul style="list-style-type: none"> -
Petri-Nets	<ul style="list-style-type: none"> - State machines - Marked graphs - Timed Petri Nets 	<ul style="list-style-type: none"> - Appropriate for distributed applications - Well-known theory for formally proving properties 	<ul style="list-style-type: none"> - No programming elements - No hierarchy - Problems with modeling timing
Discrete event based	<ul style="list-style-type: none"> - VHDL - SystemC - Verilog - SpecC 	<ul style="list-style-type: none"> - Time is an integral part of a discrete-event model 	<ul style="list-style-type: none"> - It can be expensive: sorting time stamps can be time consuming - Less efficient when the activity of a system increases, because of the overhead of simultaneous events
Von-Neumann	<ul style="list-style-type: none"> - ADA - C - Fortran - Java 	<ul style="list-style-type: none"> - Flexibility - Enable economical use of fast memory 	<ul style="list-style-type: none"> - Operation destroy information and this could be dangerous (data and instruction share the same memory) - Execution process is slower
Synchronous reactive	<ul style="list-style-type: none"> - Esterel - Lustre - LabVIEW 	<ul style="list-style-type: none"> - It involves synchrony - Good at managing sporadic data (where the absence of events has meaning) 	<ul style="list-style-type: none"> - High cost (in area and power consumption) of precise clock distribution - Synchrony unfeasible when the clock speed and circuit size become large

Table 2.3: Examples of architectural languages

Language	Description	Origin	Notation	Code generation	Free tools	Hierarchies	PROs	CONs	Last known activity
AADL	Developed by SAE and derived from MetaH.	Industry Academy	graphic textual xml	YES	YES	YES	- It covers different layers of the system (hw, middleware,...) - Formal verification - It can model the target	It does not model clocks	2021
ABACUS	Developed by the University of Technology of Sydney.	Academy	graphic textual xml	YES	NO	YES	It supports all of the major solutions (UML, SysML,...)	Not sufficient for the complete development of a system.	2021
ACME	Designed at Carnegie Mellon University it is a generic ADL.	Academy	graphic textual	NO	YES	YES	It provides a common intermediate representation for various tools	Dynamic aspects of architectural evolution.	2018
AUTOSAR	Worldwide partnership of vehicle manufacturers for a standardized sw architecture.	Industry	graphic textual	YES	YES	YES	Scalability to different vehicles and variants	Specific for automotive systems	2021
ByADL	Designed at University of L'Aquila, on top of DUALLy.	Academy	graphic textual	YES	NO	YES	-	-	2012
Darwin	Designed at Imperial College of London, it allows a component- or object-based approach.	Academy	graphic textual	NO	YES	YES	It supports the structure of parallel programs and modeling of network topologies	Lack of management of FT aspects.	2013
EAST-ADL	ADL for Automotive ESs, it complements AUTOSAR.	Industry	graphic textual	YES	YES	YES	High abstraction layer	Specific for automotive systems	2021
MetaH	Developed by Honeywell, basis for AADL.	Industry	graphic textual	YES	YES	YES	- Strong component semantics - Non-functional specification	Not largely deployed	2005
Rapide	Designed at Stanford It adopts a new event-based execution model.	Academy	graphic textual	YES	YES	YES	Modeling of component interfaces and externally visible behavior	Not sufficient for the complete development of a system.	2005
SADL	Developed at the National Aeronautics and Space Administration's Marshall Space Flight Center	Industry	graphic textual	NO	YES	YES	It supports the specification of non-functional properties	No precise relationship between a graphical description and the underlying semantic model	N.A.
StratusML	Developed at the University of Waterloo (Canada) for modeling cloud applications..	Academy	graphic textual	YES	YES	NO	- Flexibility, re-usability - It maintains consistency between the artifacts of cloud applications	Non-functional specifications	2018
SysML	Developed as extension of UML by INCOSE and OMG	Industry	graphic textual xml	YES	YES	YES	- It can produce specifications for heterogeneous teams. - It regroups structural, functional and behavioural views of a system	Not sufficient for the complete development of a system	2021
UML/Marte	It extends UML for modeling real-time systems	Industry	graphic textual	YES	YES	YES	- Common way to model hw/sw - Interoperability between tools	Concept of time is ambiguous	2021
UML-RT	Extension of UML to deal with timing properties	Industry	graphic	NO	NO	YES	- Wide range of diagrams - Large number of free tools - Real time aspects	Low degree of formal verification	2021
Unicon	General purpose ADL of Carnegie Mellon University.	Academy	graphic textual	YES	YES	YES	Emphasis on generation of connectors	No evolution support	1999
xADL	Developed by the University of California, it is defined as a set of XML schemas.	Academy	xml	YES	YES	YES	Extensibility and flexibility	XML schemas do not provide facilities to define the semantics of individual elements.	2021

2.3.2 Formal Methods for Specification

At the heart of FM one finds the notion of specification. A specification is a model of a system that contains a description of its desired behavior: *what* is to be implemented (by opposition to *how*). The definition of a specification and the analysis of its behavior may be carried out formally. A specification essentially describes the manipulated data and how they evolve, i.e., the operations that transform them. Sommerville [49] divided specification styles for complex systems in model-based and algebraic approaches. The two main approaches to formal specification differ on the focus given to these two aspects:

- The behavior of the modelled system can be expressed by focusing on its *operations*, available mechanisms (services), or actions that can be performed. In this view the key element is a clear definition of the modifications or changes performed by each operation on the internal state of the modelled system. Such specification languages are referred to as state-based or *model-based* specification languages.
- On the other hand, the behavior of the target system can be expressed by focusing on the manipulated data, how they evolve, or the way in which they are related. This class of specifications includes *algebraic specifications*, sometimes also known as axiomatic specifications.

For a model-based specification, a system is defined in terms of mathematical entities (e.g., sets, relations, sequences), giving the explicit specification of abstract machines. Operations are defined in terms of abstract states and how they affect those entities. The algebraic style specifies abstract data types in terms of axioms defining relationships between its operations (this style is sometimes also denoted as property-oriented specification).

2.3.3 Formal Methods for Verification

Rather than simply constructing specifications and models, one is interested in proving properties about them. We are in the realm of formal verification. Proving properties about specifications presupposes the use of some logical system.

Table 2.4: Truth table

INPUT		OUTPUT		
x	y	NOT ($\neg x$)	OR ($x \vee y$)	AND ($x \wedge y$)
0	0	1	0	0
0	1		1	0
1	0	0	1	0
1	1		1	1

Hints of Boolean Algebra

Boolean algebra was introduced in 1938 by Shannon to deal with problems that had arisen in the design of relay switching circuits. It allows the concise description and manipulation of binary variables. Variables in Boolean algebra may assume only one of two possible values. Boolean algebra operates with three functional operators (aka logical connectives): "NOT", "OR", and "AND". These building blocks are comparable to taking the negative, adding, and multiplying in ordinary algebra. The possible input and output combinations can be arranged in tabular form, called a *Truth Table* (see Table 2.4). *Boolean formulae* are built over:

- the two constants 0 (false) and 1 (true)
- a denumerable set of variables
- the logical connectives NOT, OR, and AND

A *literal* is either a boolean variable v or its negation $\neg v$. If v is a positive literal, $\neg v$ is a negative literal; they are opposite. The *opposite* of a literal v is $\neg v$, and $\neg(\neg v) = v$. A *product* is a set of literals that does not contain both a literal and its opposite. A product is assimilated with the conjunction of its elements. The *order* of a product π is the number of its literals and is denoted by $|\pi|$. An *assignment* over v is any mapping from v to $\{0,1\}$. Let F and G be two formulas. If any assignment satisfying F satisfies G as well, then F *implies* G , which is denoted by \models .

Canonical form Let be V a finite set of variables. A product that contains a literal built over each variable of V is a *minterm* of V , i.e., it is a product (AND) of the n variables in which each variable is complemented if the value assigned to it is 0, and uncomplemented if it is 1. Any boolean function can be expressed as a sum (OR) of its 1-minterms. Dually, a *maxterm* is a sum (OR)

of the n variables (literals) in which each variable is complemented if the value assigned to it is 1, and uncomplemented if it is 0. Any boolean function that is expressed as a product of maxterms, aka CNF, or as a sum of minterms, aka *Disjunctive Normal Form* (DNF), is said to be in its *canonical form*. It is often convenient to consider boolean formulas as sets of minterms and therefore to consider logical operations as set operations. The formula $\neg F$ corresponds to the complement of the set of minterms that corresponds to F . The formulas $F \wedge G$ and $F \vee G$ correspond to the intersection and the union, respectively, of the sets of minterms that correspond to F and G . A formula F is monotone if for any minterm $\neg v \wedge \pi$ that satisfies F , the minterm $v \wedge \pi$ satisfies F as well.

Standard form Canonical form means that each term of boolean functions contains all input variables either in true form or complemented form. But each term may have any number of variables. A boolean function is in its *standard form* if there exists at least one term that does not contain all variables. Standard forms require fewer operators than canonical forms. The standard expressions are in either:

- *Sum of Products* (SOP) form: expressions that contain AND terms, called product terms or *implicants*. E.g., $F = xy + \neg xyz + x\neg y$.
- *Product of Sums* (POS): expressions that contain OR terms, called sum terms or *implicates*. E.g., $F = (\neg x + \neg xy)(x + \neg y + \neg z)(\neg x + y + \neg z)$.

There can be several different sums of products and products of sums for a given function, i.e., they are not unique. When each product term in a sum of products form cannot be reduced any further, it is called *Prime Implicant*. Dually, when each sum term in a product of sums form cannot be reduced any further, it is called *Prime implicates*.

Logic in a Nutshell

Whichever method is used, proving properties about specifications presupposes the use of some logical system. Logic can be described as the study of the principles of reasoning. We are interested in doing this formally, so that the arguments are valid and can be defended rigorously. *Symbolic logic* is the branch of mathematics devoted to formal logic, i.e., to the study of logical languages, their semantics, their proof theory, and the way in which these are related. A logic consists of:

- A *logical language* in which sentences are expressed. A logical language is a formal language having a precise, syntactic characterization of well-formed sentences. A logical language consists of logical symbols, characterized by having a fixed interpretation, and non-logical ones, whose interpretations are not fixed. These symbols are combined together to compose well-formed formulae.
- A *semantics* that differentiates valid sentences from refutable ones. The semantics is defined in terms of the truth values of sentences. This is done using an interpretation function that assigns meaning to the basic components, given some domain of objects that our reasoning is concerned with.
- An *inference system* (or proof system) that supports the formalization of arguments justifying the validity of sentences. The inference system is composed of a set of axioms (sentences of the logic that are accepted as true) and inference rules (that give ways of deriving the right conclusions from a given set of premises).

Three well-known logics are the following:

- **Propositional logic** (also known as the propositional calculus) is the simplest logic available. The formulae of propositional logic are built from atomic propositions, which are sentences with no internal structure and which one can classify as being "true" or "false". Propositions are declarative sentences such as "Antonio is the father of Romuald" or " $3 < 5$ ". Propositions are combined using Boolean operators that capture notions like "not", "and", "or", "implies", etc. Propositional logic is the study of the way in which the truth of one statement affects that of another. However, the expressiveness of propositional logic is very limited. For instance, let us try to describe the logical content of the following sentence:

For all numbers x and y , if x is greater equal zero and y is greater equal zero, then x times y is zero or not less than x .

In propositional logic, we can write the following formula:

$$a \wedge b \implies c \vee \neg d$$

where we use the propositional variables a , b , c , and d as abstractions of the sentences “ x is greater equal zero”, “ y is greater equal zero”, “ x times y is zero”, and “ x times y is less than x ”, respectively. But we completely ignore the sentence “for all numbers x and y ”. While the original sentence is true for arbitrary numbers x and y , the formula can be true or false, depending on the truth values of the propositional variables. This is because the formula does not describe the content of the sentence. More in general, we cannot use propositional logic to reason about propositions that obey laws (such as arithmetic laws) that go beyond the logical inference system. Propositional logic is not expressive enough to talk about concrete objects like numbers, their relationships, and the fact whether a sentence is true for all or just for just some objects of a domain.

- **First-order predicate logic**, also called *First-order predicate logic* (FOL) or predicate logic, is a richer logic than propositional logic. In addition to the symbols of propositional logic, it contains elements that allow us to reason about individuals of a given domain of discourse. These include functions, predicates, and quantification over individuals, dealing with the notions of “there exists” and “for all”. There are two sorts of things involved in a FOL formula: *terms*, which denote objects; and *formulae*, which are interpreted as truth values. FOL is also known as the *predicate calculus* in the sense that it is a calculus for reasoning about predicates such as “ x is the father of y ” or “ $x < x + 1$ ”. While propositions are either true or false, predicates evaluate to true or false depending on the values given to their parameters (x and y in the previous examples). Quantification over individuals makes possible to express concepts such as “every person has a father”. A term can be one of the following:
 - A *variable* v : to a variable we may assign varying objects.
 - A *constant* c : to a constant we assign a fixed object.
 - A *function application* f : the application of a function symbol f with arity $n \geq 1$ to a sequence of n terms $t_1; \dots; t_n$. Such a function symbol denotes an n -ary function that, when applied to n objects, returns another such object.

Apart from the constructions already present in propositional logic, a formula can be one of the following entities:

- An *atomic predicate* p : the application of a predicate symbol p with arity $n \geq 1$ to a sequence of n terms $t_1; \dots; t_n$. Such a predicate symbol denotes an n -ary predicate that, when applied to n objects, returns "true" or "false".
- A *universally quantified formula* $(\forall v : F)$: you read "for all (possible objects assigned to) v , F is true", with the *universal quantifier* "for all" (\forall) applied to a variable v and a formula F .
- A *existentially quantified formula* $(\exists v : F)$: you read "there exists some (possible objects assigned to) v , for which F is true", with the *existential quantifier* "exists" (\exists) applied to a variable v and a formula F .

Please note that the difference between a function and a predicate is that the application of a function returns an object, while the application of a predicate returns a truth value. Now we are able to write the sentence introduced in the previous paragraph:

For all numbers x and y , if x is greater equal zero and y is greater equal zero, then x times y is zero or not less than x .

In FOL, we can write the formula as follows:

$$\forall x : \forall y : \text{greaterEqual}(x, \text{zero}) \wedge \text{greaterEqual}(y, \text{zero}) \implies \\ \text{equal}(\text{times}(x, y), \text{zero}) \vee \neg \text{lessThan}(\text{times}(x, y), x)$$

where x and y are variables, zero is a constant, times is a binary function, and greaterEqual , equal , and lessThan are binary predicates.

- **High-order logic** is distinguished from FOL in several ways. It has both individual and relational variables, and both types of variables can be quantified, so quantifiers may apply to variables standing for predicates. Predicates can take as arguments both individual symbols and predicate symbols (these are higher-order predicates). Concepts such as "every property that holds for x also holds for y " can be naturally expressed in higher-order logic. First-order logic is more expressive than propositional logic: it has more rules, that allow one to construct more complex formulae. In turn, higher-order logic is more expressive than FOL.

- **Temporal Logic** In the examples we gave, the truth value of the statements are static and cannot vary over time, i.e., they are always true or always false. Consider the statement: "It is raining". Although the meaning of this statement is stable over time, its truth value can vary, sometimes it is true and sometimes it is false. Time can be considered as an object of discourse, making statements depend on a time variable, but the result is very clumsy. "Temporal logics" is used to refer to logics that implicitly include a notion of time, providing a way to represent temporal information in the logical framework. In temporal logics the truth of a formula is not fixed in the semantics but depends on the point in time in which it is considered. Temporal logics have two kinds of operators: the usual *logical connectives* (such as "not", "and" or "implies") and *temporal connectives* (such as "eventually", "always" or "until"), allowing one to express statements like "It will eventually rain". There exist many different sorts of temporal logics. Concerning the way time is viewed, they are classified as linear-time when time is represented by a sequence of time instants, referred to as *Linear Temporal Logic* (LTL), or branching-time, when time is viewed as a tree of time instants, having the present instant as root, and with each branch corresponding to one possible future evolution, referred to as *Computation tree logic* (CTL). Temporal logics have found important applications in FM and in modeling behavioral aspects.

Proof Tools

We can arrange proof tools in two families, presented below, which reflect the trade-off between expressiveness and automation.

- **Automated Theorem Provers** Theorem proving uses mathematical logic to formulate a theorem about the correctness of a design using first-order logic, then a general purpose theorem-prover is used to construct the proof. This step is automatic, once the proof engine has been adequately parameterized. Unlike model checkers (covered below), theorem provers may be able to employ techniques that allow for reasoning. Second and higher-order logics are more expressive, and thus can be used to model more complex systems. However, user interaction is required to guide the proof tools. Hence, the disadvantage of theorem proving is that it cannot be fully automated.

- **Proof Assistants** employ highly expressive (and thus undecidable) underlying logics, such as higher-order logic, as many properties need its power and elegance to be adequately expressed and proved. Proof assistants typically combine the following two modules:
 - a proof-checker, responsible for verifying the well-formedness of the theories defined in the modeling process, and for checking the correctness of proofs;
 - an interactive proof development system, to help users developing proofs. When the construction of a proof is finished, a proof script can be stored, describing that construction.

In most proof assistants, proofs are interactively constructed by applying high-level proof-manipulation functions, usually known as *tactics*. Each tactic encodes a proof step. A proof of a property ϕ is established by applying (in an appropriate order and with the right parameters) a set of tactics, in order to construct a proof tree linking axioms and theorems to the conclusion ϕ .

Model Checking

Model checking is a technique for the verification of finite-state (concurrent) systems (typically modelled by automata). It is one of the most widely used families of FM tools. The idea of model checking is that the expected properties of the model are expressed by formulae of a temporal logic, and efficient symbolic algorithms are used to traverse the model in its entirety, so as to verify if all possible configurations validate those properties. The set of all states is called *state space* of the model. When a system possesses a finite state space, model-checking algorithms may in theory be used to realize the automatic demonstration of properties. If a property is not valid, a counterexample is exhibited. A serious drawback of this approach is state space explosion: the transition graph typically grows exponentially on the size of the system, with the immediate consequence that no matter how efficient the checking algorithms are, the exploration of the state space eventually becomes impracticable. Different techniques have been proposed that try to solve this problem. Abstraction is one such technique: a simplified version of the model is proposed, called an abstract model, whose state space may be explored within reasonable time. A major breakthrough was enabled by the introduction of

symbolic model checking [50]: the basic idea is to manipulate sets of states and transitions, using a logical formalism to represent the characteristic functions of such sets. Since a small logical formula may admit a large number of models, this results in many practical cases in a very compact representation which can be effectively manipulated.

2.3.4 Formal Methods for Implementation

FM can be also employed to guarantee that an implementation has the same behavior as the specification. The problem we want to address is the following: given a specification that enjoys the desired properties, how to obtain an implementation whose behavior matches the specification? The solutions fit in two categories:

- either the specification is itself a program that can be directly executed, or
- an implementation is produced from the specification, in which case the problem of the *correctness* of derivations must be dealt with.

One approach for dealing with this problem focuses on the derivation mechanisms, which can be restricted in appropriate ways, to ensure that the derived code satisfies the properties of the original specification. The second approach is to make the (not necessarily correct) derivation process generate a set of proof obligations such that, if all these obligations can be proved, this guarantees the correctness of the implementation with respect to the specification. Both these approaches are sometimes referred to as *correct-by-construction* software development.

2.3.5 How to Choose a Formal Method

Due to recent trends, there is a huge number of FM, languages, and available tools, each developed for a particular domain and purpose. Currently, there are more than 100 different FM listed on the de-facto FM repository at the World Wide Web Virtual Library on FM [51]. This is a de-facto database of anything relating to FM, with entries mostly from the USA and Europe. FM are in different stages of development, in a wide spectrum from formal languages with no tool support, to internationally standardized languages with tool support and industrial users. The field of FM is evolving rapidly, making inroads

into industrial practice. Companies that have safety, security, compliance to (often international) standards, certification, or product quality in mind are interested in FM. Which formal method is most suitable for a specific project is dependent on the kind of system and the kind of properties to be proved.

System classification

According to Schneider [8], systems can be classified as follows, based on their architecture:

- Asynchronous or synchronous hardware
- Analogue or digital hardware
- Mono- or multi-processor systems
- Imperative/functional/logic-based/object-oriented software
- Multi-threaded or sequential software
- Conventional or real-time operating systems
- ES or local systems or distributed systems

As specific architectures such as the ones given above are dedicated to specific tasks, it is clear that one formal method will be more or less suited than the other. The author discusses some of the distinguishing characteristics of these systems and what FM are most appropriate for them. For example, multi-threaded software might require the implementation of complex control tasks and therefore requires a high-level description language. In sequential systems one could use logic-based languages and functional languages. Furthermore, also based on Schneider work [8], systems can be classified based on the type of interaction:

- *Transformational systems.* Systems that read some input data and produce output. As the output is produced at termination, these systems should always terminate. A compiler is an example of such a system.
- *Interactive systems.* Systems that continuously run and interact with the environment. When the environment gives an action, the system replies with a reaction. The environment has to wait until the system is ready for new actions.

- *Reactive systems*. Similar to interactive systems, only that the environment can freely decide when to start new actions. Therefore the system has to react to a given action before the next action comes. This kind of system falls under the real-time systems category.

Considering this last classification, it is reactive systems that are the most challenging to implement.

System properties

Many formalisms can be classified by which system properties they can express. A taxonomy of properties is given next, based on Schneider work [8], without taking a specific formalism into account.

- *Safety properties* state that for all computations of the system, and for all instances of time, some property will invariantly hold, i.e: "something bad will never happen".
- *Liveness properties* state that some desired state of the system can eventually be reached, i.e: "something good will eventually happen".
- *Fairness properties* state that some property will infinitely often hold, i.e: "some property will infinitely often hold".
- *Persistence properties* are related to the stabilization of certain properties, i.e: "stabilization of certain properties". In general, a persistence property describes that for all possible computations, there is a point of time when a certain property will always hold.

Summary

As can be seen, a plethora of formalisms has been proposed. They have both advantages and disadvantages. Hence, the complexity of the decision procedures should be taken with care as an argument for or against a formalism. Anyway, it is important to have different ways to specify a particular property because one can be more suited than the other, and you can employ whichever one best suits the problem.

2.4 System Optimization

Many combinatorial satisfiability and optimization problems can be formulated as a *Constraint Satisfaction Problem* (CSP) [52]: the problem is to find a variable assignment to all variables that satisfies all hard constraints and at the same time optimizes a global cost function for the soft constraints. Such an optimization problem is sometimes also called *Constraint Optimization Problem* (COP) [53]. Optimization problems can be classified based on:

- The type of constraints: unconstrained or constrained.
- Nature of the equations involved: linear or non-linear.
- Admissible value of the decision variables: discrete or continuous.
- Deterministic nature of the variables: deterministic or stochastic.
- Number of objective functions: single-objective or multi-objectives.

Below we give a brief overview over different technologies in problem optimization, discriminating against the number of objective functions.

2.4.1 Single-objective Optimization

A COP is essentially an optimization problem that consists of discrete decision variables and finite domain search [54]. In the following we focus on the techniques used for solving single-objective COPs.

Branch and Bound

Branch and bound [55] is an exact algorithm that is used to solve optimization problems to retrieve globally optimum solution. The design space is explored by building a tree, where the root of the tree represents the problem and the leaf nodes represent solutions to the problem. Internal nodes of the tree represent sub-problems, such that the size of the sub-problem reduces as we move from the root to the leaf. The tree is dynamically construct by iteratively branching and pruning. The branching strategy divides the problem into two mutually exclusive sub-problems that can in turn be divided into smaller sub-problems. Many branching strategies can be applied such as *breadth-first*, *depth-first* and *best-first*. The pruning strategy is used to prune out sub-problems that will not lead to optimal solutions. This is done by computing a lower bound of the

sub-problems. If the lower bound is greater than the best solution so far then the sub-problem can be pruned out. This method has been used to solve – for instance - scheduling problems [56]. Branch and bound algorithms have been used to solve single-objective optimization problems formulated as a CSP or a linear program.

Mathematical Programming

Mathematical Programming [57] facilitates the modeling and solution of a broad class of COPs. A commonly used model in mathematical programming is the *Linear Programming* (LP). A linear program is an optimization problem that seeks to minimize a cost function subject to a set of linear constraints. A linear program in its standard form can be formulated as:

$$\begin{aligned}
 & \text{Minimize: } c^T x \\
 & \text{Subject to: } Ax \leq b \\
 & \qquad \qquad \qquad x \geq 0
 \end{aligned}
 \tag{2.1}$$

where A is a matrix, and c , b are vectors of known coefficients. A , c and b are given and x represents a vector of decision variable whose value is to be determined. The objective function that combines the different variables to express a goal has to be maximized or minimized. LP has proved efficient in solving a variety of exploration problems like scheduling [58] and hardware-software co-design [59]. An *Integer Linear Programming* (ILP) extends the concepts of linear programming by adding integrality constraints to the linear programs. A *Mixed Integer Linear Programming* (MILP) relaxes the integrality constraint of ILP and can have variables that have one or more variables which take real values. For instance, MILP has been used for modeling hardware/software partitioning [59], network design [60], real-time scheduling [61], electric power systems [62], and wireless networks [63]. The following search methods have been used for solving LP models:

- *Simplex Method* [64] is the most popular solution technique for solving linear programs developed in 1947. However, Simplex method has an exponential time complexity. Karmakar's *projective scaling method* [65] is a polynomial time algorithm for solving linear programs.

- *Branch and bound* [55] is used in combination with linear relaxations to solve ILP's. Using branch and bound an ILP solver decomposes the problem into smaller sub-problems recursively. The integrality constraints are replaced with lower and upper bounds on the variables, thus transformation it into an LP problem. The bound obtained by LP relaxation is used to discard sub-problems that have a bound than the best known solution.
- *Metaheuristics: Local Search* (LS) algorithms have been used to solve large instances of ILP problems [66].

Local Search

LS algorithms are one of the most widely and successfully applied approximate algorithms. A LS algorithm starts with a given solution of the problem and iteratively tries to find a better solution in the neighborhood of the current solution. In case a better solution is found it replaces the current solution. This step is iteratively performed until no better solution can be found in the neighborhood. The disadvantage of using the LS algorithm is that it can get stuck at a local minima. In order to overcome this drawback several improvements have been proposed. One approach is to restart LS from a new, randomly selected solution. This approach is applied in *Iterated Local Search* (ILS) [67] and *Greedy Randomized Adaptive Search Procedure* (GRASP) [68]. Another possible approach is to accept worse solutions, thus escaping the local optima. This approach is applied in *Simulated Annealing* (SA) [69] and *Tabu Search* (TS) [70]. Kirkpatrick [71] applied the annealing concept from physics to solving COPs. Annealing is based on the principle of mechanics whereby a substance is heated and then slowly cooled to get a strong crystalline structure. If the substance is not heated to the right temperature or the cooling is too quick then the process, then the resulting solid is weak and brittle. Thus, it is imperative that the cooling is done at the right rate.

- SA algorithm simulates the energy changes in the system till it converges to an equilibrium state. The optimization problem is analogous to the system, where the system state represents a solution and the energy of the system represents the objective function. Starting from an initial solution, SA incorporates significant randomization while traversing the

state space. In each iteration a random neighbor is selected. If the objective value of the neighbor is better the move is always accepted. If not, then the move is accepted with a probability that is a function of the current temperature (a control parameter) and the difference in the objective value. Initially the temperature is high and the probability of accepting non improving solutions is also high, but as the simulated temperature decreases according to the cooling schedule, fewer such moves are accepted. The search stops after stopping condition is met, typically when the probability becomes negligible or the temperature reaches a certain threshold. Different versions of SA have been used for design space exploration, for example Gupta and Bic used a parallelized SA algorithm [72], where a number of instances of the algorithm are run in parallel.

- TS algorithm [70], like SA, also accepts non-improving solutions to escape from the local optima. In contrast to random moves used in SA, the neighborhood is explored in a deterministic manner in TS. Like the basic LS algorithm, a better neighbor replaces the current solution, but the search continues even after reaching the local optima by accepting non-improving solutions. This can lead to cycles if the solution accepted has been previously traversed. TS avoids cycles by maintaining a list of recently visited solutions/moves (tabu list) and discards all those neighbors present in the list. TS also uses certain conditions, called aspiration criteria to accept tabu moves if it generates a better solution among the set of solutions possessing a given attribute. Advanced techniques like medium-term memory and long-term memories are commonly used to handle intensification and diversification of the search. TS has also been used for DSE of ES [73].

Compared to multi-objective optimization, single-objective optimization problems are easier to solve because of the total ordering of the solutions in the search space. In general exact techniques perform better for tightly constrained problems, while approximate algorithms perform better for unconstrained optimization problems. However, approximate techniques are mostly used when exact techniques are unable to provide solution in acceptable amount of time. This is true for large instances of COPs. The approximate algorithms are generally used where reasonably good solutions obtained in polynomial time are preferred over globally optimum solutions.

2.4.2 Multi-objective Optimization

Problems involving multiple conflicting objectives that have to be considered simultaneously are generally known as *Multiple Criteria Decision Making* (MCDM) problems. The design of those systems is challenging because engineers have to deal with a large number of non-functional or quality requirements such as safety, availability, reliability, maintainability and temporal correctness requirements. One major difficulty is that these non-functional requirements conflict with one another. To construct a system that fulfils all its quality requirements is often not possible. As a consequence, system engineers have to consider several design alternatives and identify a solution that fulfills most quality objectives. This process is called *trade-off* analysis. It is a cost and skill intensive task whose intention is to find a set of architecture specifications that solve a MOOP, where the objectives represent different quality attributes. The method involved in the decision process could be (see Figure 2.20):

- **Exact:** essentially combine the multiple objectives into one single objective. Well known exact methods are weighted sum scalarization, compromise solution, goal programming, branch and bound.
- **Approximation:** heuristics that seek near-optimal solutions at a reasonable cost and meta-heuristics that guide and modify the operation of subordinate heuristics by combining intelligently different concepts for exploring and exploiting the search space. Approximation methods include, but are not limited to, Constraint LP, *Genetic Algorithms* (GA), *Evolutionary Algorithms* (EA), *Neural Network* (NN), SA, TS, GRASP, LS.

In the context of MOOPs, it is usual to distinguish the methods following the role of the *Decision Maker* (DM) in the resolution process. The DM is a person who is assumed to know the problem considered and be able to provide preference information related to the objectives and/or different solutions in some form. Information provided by the decision maker often concerns his preferences. Miettinen [74] groups methods in non-interactive and interactive classes. Non-interactive methods are those where either no DM takes part in the solution process or (s)he expresses preference relations before or after the process. In a priori mode DM first articulates preference information and aspirations and then the solution process tries to find an optimal solution satisfying

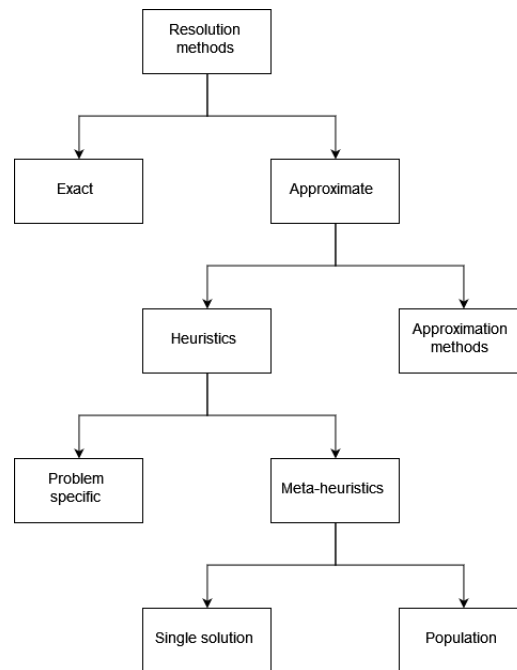


Figure 2.20: Resolution methods of MOOPs

them as well as possible. This is a straightforward approach but the difficulty is that the DM does not necessarily know the possibilities and limitations of the problem beforehand and may have too optimistic or pessimistic expectations. In a posteriori methods, a representation of the set of optimal solutions is first generated and then the DM is supposed to select the most preferred one among them. This approach gives the DM an overview of different solutions available but if there are more than two objectives in the problem, it may be difficult for the DM to analyze the large amount of information. Sometimes, there is no DM and in those cases we must use so-called no-preference methods. In interactive approaches, an iterative solution algorithm (which can be called a solution pattern) is formed and repeated (typically several times). After each iteration, some information is given to the DM and (s)he is asked to specify preference information (in the form that the method in question can utilize).

Linear Programming

Coming from classical Operations Research domain, LP creates quantitative models to support decisions for advanced optimization problems. Many practical problems can be expressed as LP problems. Although the number of possible decisions is vast, the theory behind LP drastically reduces the number of possible solutions that must be checked. More in details, LP problems

can be classified as follows:

- LP: it is a technique for the optimization of a linear objective function, subject to linear equality and linear inequality constraints (e.g., solved by Simplex [75]). Special cases exist, like stochastic linear programming [76].
- ILP (integer LP): the objective function and the constraints are linear (e.g., Branch and Bound [77]).
- MILP (mixed ILP): some decision variables are not discrete [78]
- *Multi-Objective Pseudo-Boolean ILP* (MO PB-ILP): the ILP problem has binary variables and multiple conflicting objectives [79]

Meta-heuristic Search Algorithms

A meta-heuristic is to automatically determine an appropriate heuristic to find a non-optimal, but sufficiently good solution for a given search problem, also usable to heuristically solve an optimization problem. References on the theory and applications of meta-heuristics can be found in literature [80], [81]. The benefit prior to an exact optimal algorithm is that the computational complexity is reduced, such the search can be performed more efficient in less time. This is especially relevant for NP-hard problems. Examples are:

- SA [71] is a probabilistic technique to approximate a global optimum in large search spaces. Inspired by the technique of annealing in metallurgy (applying slow cooling), SA algorithms slowly decrease the probability of accepting worse solutions, while exploring the solution space. Local optima can be forsaken in order to find other better optima.
- TS [82], [83] guides a local heuristic neighborhood search procedure to explore solutions beyond a local optimum, by using a so called tabu list. Hence, TS is able to cross boundaries of feasibility or local optimality.
- EA, [84], also used for multi-objective optimization [85], for instance implemented in the *Multi-Objective Evolutionary Algorithms* (MOEA) Framework. GA [86] belongs to the larger class EA.
- Swarm intelligence approaches, like:

Particle Swarm Optimization (PSO) [87]: a concept for the optimization of continuous non-linear functions, having ties to both GA and EA.

Ant Colony Optimization (ACO) [88]: a probabilistic technique, inspired by the behavior of real ants, which can be reduced to finding good paths through graphs.

Metaheuristics can be further distinguished into algorithms applying:

- Local (neighborhood) search strategies (e.g., TS and
- Global search strategies (e.g., GA and PSO.
- Combination of both: the SA approach mediates between local and global search.

Answer Set Programming

Answer set programming (ASP) describes a problem as a logic program, a set of axioms, and a goal statement, under the answer set (stable model) semantics of logic programming [89] in such a way that the models of the program (answer sets) correspond to the solutions of the problem [90].

Satisfiability

SATisfiability (SAT) [91] is the problem of determining if a formula expressing a constraint has a model (i.e. a solution). Given a propositional formula on a set of Boolean variables, a *SATisfiability* (SAT) solver determines if there exists an assignment of the variables such that the formula evaluates to "true" or proves that no such assignment exists. A large number of problems can be described in terms of satisfiability, including planning, scheduling, software and hardware verification, and optimization. Many of these problems can be encoded by boolean formulas and solved using Boolean SAT solvers.

Satisfiability Modulo Theories

Other problems require the added expressiveness of equality, uninterpreted function symbols, arithmetic, arrays, datatype operations, and quantifiers. Such problems can be handled by solvers for theory satisfiability or *Satisfiability Modulo Theories* (SMT) [92]. SMT is the problem of deciding the satisfiability of Boolean combinations of propositional atoms and theory atoms [93].

SMT can be seen therefore as an extension of SAT in which the input formula is expressed in (a subset of) FOL (typically without quantifiers) with respect to a background theory. Examples of useful theories are *Equality and uninterpreted functions* (EUF), difference logic and linear arithmetic (either over the reals or the integers), the theory of arrays, and bit vectors, as well as combinations of those theories. Implementations are, for instance, MathSAT, Yices, Z3 and CVC4. The dominating approach for SMT, which underlies most state-of-the-art tools, is based on the integration of a SAT solver and one or more domain-specific solvers for the background theories. The SAT solver enumerates truth assignments that satisfy the Boolean abstraction of the input formula, where distinct theory-specific subformulas are represented/abstracted by distinct Boolean atoms, whilst the domain-specific solvers check the consistency in the respective background theory of the set of literals corresponding to the assignments enumerated. This approach is called *lazy*, in contraposition to the *eager* approach, consisting in encoding an SMT formula into an equivalently satisfiable boolean formula, and on solving the result with a SAT solver.

Optimization Modulo Theories

Optimization Modulo Theories (OMT) is an extension of SMT useful if we are interested in finding not just an arbitrary satisfying assignment, but one that optimizes (minimizes/maximizes) certain criteria, i.e. that is optimal wrt. some objective functions. They allows for finding models that make a given objective optimum through a combination of SMT and optimization procedures [94], [95], [96], [97], [98]. Multiple objectives can be handled as independent box objectives or through their linear, min-max/max-min, lexicographic or Pareto fronts combination [97], [98].

2.4.3 Approaches to Design Space Exploration

Embedded systems may have competing design objectives. Therefore the architectures must be configured to meet varied requirements and, in general, *multiple design objectives*. In MOOPs, it is characteristic that no unique solution exists but a well-chosen set of mathematically equally good solutions can be identified. These solutions that best explores the trade-offs are known as non-dominated or Pareto optimal solutions. The methods and tools applied to

the estimation of non-functional properties very much depend on the particular abstraction layer and the design objectives. At system level, estimation is particularly difficult, the sub-components to be used are not designed yet and the individual tasks of the application may not be fully specified. If only a single objective needs to be taken into account in optimization, the design points are totally ordered by their objective value. Therefore, there is a single optimal design. If multiple objectives are involved, design points are only partially ordered, i.e. there is a set of incomparable, optimal solutions. They reflect the trade-offs in the design. Optimality in this case is usually defined using the concept of *Pareto-dominance*: a design point dominates another one if it is equal or better in all criteria and strictly better in at least one. In a set of design points, those are called Pareto-optimal which are not dominated by any other. Using this notion, available approaches to the exploration of design spaces can be characterized as follows:

- *Exploration by hand*: the selection of design points is done by the designer himself.
- *Exact search*: it is an exhaustive search, i.e., all design points in a specified region of the design parameters are evaluated. Well known exact methods are the *Simplex method* [64], *weighted sum scalarization* [99], *branch and bound* [55], and *goal programming* [100].
- *Approximation*: heuristics which seek near-optimal solutions at a reasonable cost. New design points are typically generated based on the information gathered so far and by defining an appropriate neighborhood function (variation operator). Approximation methods include, but are not limited to, GA [86], EA [84], NN [101], and LS [66].
- *Problem-dependent approaches*: several possibilities have been investigated so far, such as using parameter independence in order to prune the design space [102], restricting the search to promising regions [103], and composition of sub-component exploration [104].

Many of COPs, including those which we study, are NP-hard. Furthermore, adding more objectives makes the resolution even more difficult because the number of Pareto solutions may grow exponentially. Hence, exact methods do not scale for these problems. Indeed, the focus is more on designing algorithms that are able to find an approximation of the Pareto front, i.e. a

reasonably small set of good representative solutions. Approximate methods have been developed extensively since their inception in the early 1980s and they have widespread success in attacking a variety of difficult COPs where classical optimization methods have failed to be effective and efficient. As it is not straightforward to measure the quality of a given technique in generating good approximations of the Pareto front, appropriate quality indicators are needed. A quality indicator is a function that associates a quantitative measure of goodness to each approximating set. Commonly used indicators are the *Epsilon Approximation* and the *Hypervolume*. For non-deterministic algorithms (i.e., they may return different outputs on different runs, thus it is more relevant to compare the average performance), the *Attainment function* method allows to compare the performance of two algorithms graphically. In particular, MOEA have been successful in the DSE domain, but suffer from similar problems as many others stochastic optimization strategies: in the presence of design spaces only containing few feasible solutions, they spend most of their computing time in finding feasible solutions instead of optimizing feasible ones [105]. Techniques based on SAT solvers are a more recent alternative approach to solve the problem of approximating the Pareto front, by telling us whether there is an assignment of values to the decision variables that satisfies a set of constraints. The rise in efficiency of SMT solvers has produced numerous uses for them in many areas of Computer Science and Engineering. SMT solvers allow us to encode the problem as CSP. Specifically, since we are interested in finding an optimal satisfying assignment, our problem can be encoded as a COP.

Chapter 3

Related Work

In this chapter, we discuss existing related work. On the one hand, we focus on related design and analysis approaches for FT systems. On the other hand, we discuss synthesis approaches for optimized system design decisions, and their verification. There is a considerable amount of research available in the literature. In the following we discuss the most influential existing related work.

3.1 Approaches to the Design and Analysis of Fault-Tolerant Systems

A lot of research has been performed in the area of designing safety critical systems in a FT manner, for different domains, which encompass a large number of systems, from medical systems, automobiles, airport management systems, to *Building Management Systems* (BMS). Particularly, huge effort has been spent in the *avionics domain*, for instance, to design dependable Fly-by-Wire aircraft [106], or to design partitioning mechanisms to ensure fault containment and avoid fault propagation between functions that share resources in the scope of *Integrated Modular Avionics* (IMA) [107]. Also in the *automotive domain*, system reliability is becoming increasingly important [108], especially in the

context of automated and autonomous driving [109], [110]. We can find case studies also in the *railway domain* [111]. Thanks to large-scale connectivity, a variety of functionalities are now feasible in CPS. Connectivity, however, also means that CPS function in unreliable open environments, and consequently resiliency to faults (reliability), malfunctions (safety), and attacks (security) become imperative [112].

3.1.1 Design for Graceful Degradation

Embedded applications such as transportation systems, power distribution, and telecommunication are moving toward highly distributed implementations. As a result, traditional centralized approaches are being replaced by systems in which many processors collaborate to provide system functionality, spreading it across many nodes. While it may be that redundancy is the only way to satisfy stringent reliability requirements for critical functions, not every function is critical. For example, losing some fuel economy is probably preferable to a complete vehicle failure. Thus, there is room in many ES to implement graceful degradation of functionality as a way to improve dependability for non-critical functions. A *gracefully degrading* system is one in which faults are masked and only manifest themselves in a reduced level of system functionality. Design of gracefully degrading systems has traditionally been a very difficult and error-prone task. General approaches to graceful degradation are typically limited to re-implementation of the system for a number of pre-designated fall-back configurations. Nace and Koopman [113], [114] presented an approach to design gracefully degrading distributed ES separated into three steps. First, they modeled a feature model containing the super-set of all features offered by a product family. When designing a product out of the product family, those features are selected that provide the desired functionality of the product. A feature selection algorithm optimizes which features get picked to provide which functionality. In the second step, software components were selected that fulfilled the requirements of the features that were selected in the first step. The selection was done out of a library of components. In the third step, they calculated a feasible allocation (alias deployment) of the selected software components to the micro-controllers of the system. This deployment was calculated by using a bin-packing algorithm (many other algorithms could have been chosen as well). They applied functional redundancy as FT technique. Beside the calculation of deployments with redundancy in step three, the most

related part to our work is a model about sequences of hardware failures. They modeled hardware failure sequences as a *lattice*. Each vertex of the lattice represented a set of intact available hardware components (micro-controllers). It is a lattice as multiple top level vertices exist, each representing a different product of the product family. The more to the bottom of the lattice a vertex is placed, the less hardware components were intact, and quite probably the higher was the level of degradation of the system. The level of degradation was quantified by a so called *utility* value. The utility is a numeric value that represents the desirability of particular features. In order to minimize the level of degradation, the design objective was to maximize the utility of the system in the failure sequences, represented by the lattice vertices.

Shelton et al. [115], [116], [117] introduced an approach to analyze the graceful degradation of component based systems. The intention was to build implicit graceful degradation into systems, without specifying failure scenarios a priori. They distinguished two criticality levels of components (critical and non-critical), two states of components (working and failed), and two significances of functional features (primary and auxiliary). They provided a quantitative metric of the system's ability to gracefully degrade, based on the notion of utility of system elements. The utility of the whole system and its sub-systems was calculated as non-linear utility function based on a Boolean utility of atomic components. If an atomic component had a failure, its utility was 0, else its utility was 1. The utility of the composed system elements was calculated based on the utility of the composition sub-elements, according to the utility function. During the analysis, they used a discrete event simulator to inject faults to evaluate the reaction of the system to these faults and how the system gracefully degraded. They considered a fixed hardware configuration and do not considered fault detection mechanisms.

Emberson and Bate [118], [119] introduced an approach for task allocation supporting graceful degradation in distributed embedded real-time systems. The authors reused the utility function of Shelton [117] improving and integrate it into an optimization search function based on heuristic SA approach. The aim was to analyze how many hot replicas were required to ensure a certain utility function value in a fault scenario. In contrast to them, in our work we assume a fixed amount of desired redundant instances, contained in a library of FT patterns.

Trapp et al. [120] presented a framework for FT in safety critical automotive systems, applying dynamic adaption as error handling technique. They

made use of already present implicit redundancy in vehicle designs. They mentioned that for instance in typical vehicle designs, the yaw rate of a vehicle can be calculated in ten different ways without requiring any additional sensor. However, the differently calculated yaw rates may have different qualities. This means, if one sensor fails that delivers the yaw rate currently used by a specific function, another source of yaw rate can be used, but the function that uses the yaw rate then receives a less qualitative input value. To handle this, the authors modeled functions with different degradation levels. As functions contain software components, also the components have different degradation levels (also called different configurations). Compared to their contribution, in our work we introduce a more detailed formal model. In addition, we focus on explicit redundancy.

Glaß et al. [121] tackled the design of gracefully degrading mixed critical systems by focussing on a degradation-aware reliability analysis, also considering redundant deployments. The objective was to maximize the systems reliability in different degradation modes, in which different levels of functionality were provided based on the residual sets of intact resources. Instead of optimizing the reliability in the degradation modes separately in a multi-objective manner, they offered a single objective approach in which the designer could assign weights to the different degradation modes to control how much the modes influenced the objective. During design time, the degradation modes were predefined and stored into BDDs. Critical tasks could be deployed to multiple execution units. In the model, for each degradation mode they marked which resources were defect in the mode. During run-time, a dedicated reliable observer component was able to detect failures and uses the BDD data to decide which task instances had to be activated or deactivated in which execution unit in case of a resource failure. The idea behind this work was to deactivate low critical tasks to provide opportunities to keep alive high critical fail-operational tasks. They did not focus on how to detect failures, as it was not explained how the observer component did this.

Penha et al. [122] introduced a meta-modeling approach to describe architectural patterns for fail-operational, gracefully degrading systems (as part of the SafeAdapt Project). Different redundancy patterns and graceful degradation patterns were listed from literature. A so called *Fail-Operational Graceful Degradation* (FOGD) pattern was introduced and incorporated into the pattern meta-model library. However, they did not apply a metric to measure

the degree of degradation that the systems experienced in different failure scenarios. Deployment (allocation) of software to hardware was mentioned as possible future extension.

Kim et al. [123], [124] introduced a system architecture for dependable autonomous vehicles, also supporting graceful degradation in failure scenarios, by using redundancy mechanisms based on cold standby slaves, hot standby slaves, and task re-executions. They considered that the graceful degradation of vehicles should have been appropriately adjusted depending on different situations. For instance, if a vision algorithm for pedestrian detection fails due to a failure of a micro-controller, the reaction should be different when driving on a highway, compared with driving in an urban area, as pedestrian are more likely to be present in the latter case. To sum up, as type of degradation they considered the reduction of the utilization, i.e., the ratio of *Worst-Case Execution Time* (WCET) and period, of tasks on a processor by prolonging the execution period of tasks. By this, tasks were executed less often, resulting in a degradation of the quality of service. They called this adaptive resource management.

Bozzano et al. [125] used *Architecture Analysis and Design Language* (AADL) to model nominal and faulty behaviors of a system. AADL supports to model different operating modes for model entities (like devices), as well as related mode transitions. They used AADL to model degraded modes of operations and specify mode transitions by mode transition guards and mode transition effects. One focus of their work lied on a detailed specification of the operational behavior of components, particularly covering hybrid systems with continuous and discrete values. The authors introduced a dependability analysis approach for AADL models comprising degraded modes. They also analyzed the modeled system wrt. performance requirements by applying probabilistic model checking techniques. In our work we focus more on synthesizing optimal redundant architectures for different scenarios that may appear during system run-time.

Becker [126] tackled the problem of automatic deployments of mixed critical software components. He synthesized valid redundant deployments by setting up a formal system model with formal deployment constraints, expressed with linear arithmetic and logical formulas, enabling an automated calculation of valid deployments that fulfill the constraints. He also calculated communication channels between the software components, based on component port specifications. He introduced kind of systems with mixed critical

and mixed reliable functional features, having different required levels of fail-operationality, allowing to efficiently apply different levels of redundancy of software components, considering constraints for valid types of redundancy. In addition, he considered the graceful degradation of the system, failures of hardware execution units, and failures of software components. To be able to analyze the effect of failing hardware or software to the set of available functional features, he formally described the relationship between functional features and the software components, which realized these features. To decide about explicit deactivations, he established a quantitative metric to estimate the intrinsic value of software components in order to decide about the sequence in which components (and thereby features) should have been disabled when the system was not able to provide the full set of functional features anymore. He focused on an analysis on structural architecture level, without doing a behavior analysis.

3.1.2 Design for Robustness

Hamann [127] introduced an iterative DSE approach with focus on system robustness and performance. The author focused on the optimization of the robustness of ES with respect to variations of system properties. To tackle the computationally expensive analysis for inter-dependencies, he introduced a scalable stochastic method, approximating system robustness in a multi-criterion optimization problem. Redundancy or replication mechanisms are not taken into account.

Grunske [128] tackled architecture trade-off analysis for conflicting quality requirements by using an EA and multi-objective optimization strategies, based on architecture re-factorings. The approach aimed to reduce development cost and improve the quality of the system design. As case study he used a satellite system with robustness requirements and with redundancy and fault detection mechanisms, but analyzed only a simplified architecture without any redundancy. The benefit of using an EA was the scalability to be able to handle large scale problems.

Mikic-Rakic et al. [129] presented an environment for a flexible and tailorable specification, manipulation, visualization, and estimation of deployment architectures for large-scale distributed systems. They tackled the problem of deploying interacting software components to hosts. The objective

was to find a deployment that maximizes the system's availability. They defined availability as the ratio of the number of successfully completed inter-component interactions to the total number of attempted interactions over a period of time. They investigated six algorithms to increase availability by calculating new deployments, named 1) exact, 2) unbiased stochastic, 3) biased stochastic, 4) greedy, 5) clustering, 6) decentralized algorithm. Redundancy or replication was not considered by their approach.

Junker [130] introduced an artifact model to express availability requirements. The behavior of the system was modeled based on the FOCUS theory [131], enriched with availability metrics. Also a modeling guideline was introduced about how to create the availability specific model artifacts. The analysis was based on using the probabilistic model checker PRISM. Common FT techniques for reliability and robustness was the efficient use of redundancy. Various strategies and combinations of redundancy have been proposed. We are going to analyze in detail those related works in the following (see Section 3.3.2).

3.1.3 Design for Mixed Criticality

Saraswat et al. [132] enabled FT for mixed criticality multiprocessor systems by employing dynamic task migrations between processors in case of permanent processor faults. Mixed safety criticality was considered by distinguishing tasks with requirements to survive transient processor faults, permanent processor faults, or tasks with FT requirements. Also mixed time criticality was considered, distinguishing soft and hard real-time deadlines and by using *Earliest Deadline First* (EDF) scheduling for hard real-time tasks, and *Constant Bandwidth Server* (CBS) scheduling for soft real-time tasks. The migration decision was done dynamically at runtime in case of a detected permanent processor fault, using a greedy-based online heuristic. In case of decreasing resources due to permanent processor faults, performance degradation of soft real-time tasks may appear. The objective was to maximize the performance of soft real-time tasks by maximizing the probability that soft deadlines are met. Transient processor faults were handled using checkpointing and rollback recovery.

Baruah et al. [133] proposed a FM for representing mixed-criticality workloads in scheduling problems. In fact, many safety-critical ES are subject to certification problems: some systems may be required to meet multiple

sets of certification requirements, from different certification authorities. The authors argued that certification requirements in such systems give rise to new resource allocation and scheduling problems, that cannot be satisfactorily addressed using techniques from conventional scheduling theory. For this reason, they presented new techniques (reservation-based scheduling and priority-based scheduling) that can help with those problems.

Voss and Schätz [134] focused on a joint generation of schedules and deployment for mixed-criticality multicore architectures using shared memory. Their approach computes task and message schedules that are optimized with respect to a global discrete time base. As part of the solution, the approach generates an optimized assignment of tasks to computation resources (cores) concerning local memory constraints of cores and criticality constraints of tasks. The approach relies on a symbolic encoding scheme, based on a system model that is derived from the system architecture. The scheduling problem is described as a satisfiability problem using Boolean formulas and linear arithmetic constraints. A state-of-the-art SMT solver is used to compute the joint schedule and deployment for such architectures. Thekkilakattil et al. [135] ensured FT fixed priority scheduling of mixed criticality task-sets, having hard and soft deadlines, by re-executions of faulty critical tasks by replicated alternate tasks on different processing nodes, within the deadline of the original faulty task. They provided hard real-time FT guarantees for critical tasks offline (at design time), and ensure flexibility for non-critical tasks online (at runtime) by maximizing the resource utilization for non-critical tasks. They calculated the allocation (alias deployment) of tasks to processing nodes (alias execution units), and derived so called feasibility windows as well as scheduling attributes, like priorities of tasks. They applied ILP to derive the scheduling attributes.

Tamas-Selicean [136] developed methods and tools for distributed mixed-criticality real-time systems. He considered that the platform enforces separation by implementing partitioning mechanisms similar to IMA. He treated separately the optimizations at processor-level and at the communication network-level. At the processor level, he was interested to determine the schedule tables and the decomposition of tasks into redundant lower criticality tasks, such that all the applications were schedulable and the development and certification costs were minimized. He proposed SA and TS meta-heuristics to solve these optimization problems. At the communication network level, he was interested in the design optimization of TTEthernet networks used to transmit mixed-criticality messages. He proposed a TS-based metaheuristic for this

optimization problem.

Xie et al. [137] developed a functional level scheduling algorithm called *D_MHEFT* with a deadline-span-driven policy to achieve satisfactory system performance and low *Deadline missed ratio* (DMR) of multiple distributed mixed-criticality functions in heterogeneous distributed ES. The algorithm was implemented by changing up or down the system's criticality to achieve fair scheduling of functions whose criticality levels were larger than or equal to the system's criticality. Moreover, it could provide certain design guidelines to deadline certification in actual system design.

3.1.4 Design for Reconfiguration

Cansado et al. [138] introduced a formal design approach supporting structural reconfiguration and behavioral adaptation in FT systems. They presented a formal model of system configuration and reconfiguration contracts, based on a *Labeled Transition System* (LTS). For instance, this allows at predefined states to reconfigure a component by another one that implements a different behavioral interface. However, they do not consider mixed-criticality systems and do not tackle the deployment problem, and also redundancy or replication mechanisms to ensure fail-operational architectures are not in their scope.

Becker et al. [139] proposed an approach for architectural online reconfiguration in *AUTomotive Open System ARchitecture* (AUTOSAR) based automotive systems, to tackle robustness and flexibility in case of failures of system elements (like sensors). Reconfiguration models are added to typically fixed AUTOSAR models, to describe for instance alternatives for connectors between software components. To manage the connector reconfigurations at runtime in a fixed AUTOSAR architecture, a software component named "Reconfiguration" is added to the architecture and the modeled alternatives are transformed back into a merged architecture. The Reconfiguration component has all alternative connectors as input and internally routes the signals appropriately to the outputs. However, addition or removal of software components is not possible. Furthermore, they did not tackle mixed-criticality systems, redundant deployments of components, and did not apply an automatic synthesis.

There had been a lot of research about dynamic software architectures: the idea is to describe possible architectural evolution in the design phase, like addition or removal of components, ports or communication channels, that

are allowed to appear during runtime. As a consequence, there had been a corresponding growth of dynamic ADLs. Surveys and classifications over the different approaches are available [140], [141], [142], [143].

Another approach to describe architectural runtime reconfigurations are *dynamic Software Product Line (SPL)* [144]. SPLs describe a set of product variants that can be created from a common product line, based on reuse of components and specifying which components differ between product variants instantiated from a product line. The functional features of a product line, as well as the possible reconfigurations of the feature set when switching between product variants, can be designed for instance in a dynamic feature model [145]. *Orthogonal Variability Models (OVM)* [146]) and the *Variability Modeling Language (VML)* [147]) are other notations to model product line variability. In classical static SPLs, the variant decision is done at design time and the corresponding software architecture is fixed and does not change at runtime. In dynamic SPLs, switches between product variants are possible at runtime, performed by reconfigurations of the software architecture, like replacements, additions or removals of software components and connectors. Architectural adaptation mechanisms for dynamic SPLs are discussed in the work of Cetina et al. [148]. However, as the same for dynamic ADLs, there is no built-in focus on ensuring FT and fail-operational requirements by using appropriate redundancy.

3.1.5 Design for Self-x

After the investigation of dynamic ADLs in the late 1990s and early 2000s, a new research field opened towards dynamic systems with self-x (alias self-*) properties. A lot of approaches exist to apply self-x techniques to create dependable ES, and a lot of self-x properties have been defined: self-configuration, self-adaptation, self-organization, self-repairing, self-managing, self-optimizing, etc. The works of Salehie and Tahvildar [149], Rodosek et al. [150], Kephart and Chess [151], and Mühl et al. [152] provided good starting points to get an overview over all the mentioned self-x properties and their relationships, which we do not completely list here. We briefly overview the self-configuration property. *Self-Configuration* establishes an automated configuration of components and systems following high level policies. The rest of the system adjusts automatically and seamlessly [151]. One approach to enable self-configuration was for instance presented by Stein et al. [153]. The

synthesis that we introduce in this work is also a kind of automated configuration. However, the major use case that we consider is the application at design time. The analysis results can be stored and used to construct pre-configured dynamism, or to analyze the decision space of decision mechanisms implemented in the system. We do not focus on self-configuration at runtime by the system itself, but when using a more efficient heuristic calculation of solutions, our analysis could also be performed at runtime, contributing to establish self-configuration.

3.2 Formal Reliability Analysis of Redundant Architectures

This paper is based on the work proposed by Bozzano et al. [154] in which reliability analysis was completely automated, employing the language SMV [155] and its associated tool XSAP [156].

Other techniques that have also been quite successful for reliability analysis are based on Monte Carlo simulations [157], although simulation-based approaches do not provide exhaustive evaluation of the system. Indeed, this approach implies the analysis of a large number of samples, but can never be termed as 100% accurate. Furthermore, simulation requires an enormous amount of CPU usage.

Conversely, FM are capable of conducting precise system analysis and overcome the limitation of simulation-based methods. The idea behind formal analysis of a system is to build a mathematical model of the given system and formally verify that it meets the specifications of intended behavior. Some techniques widely used in industry to analyze redundant architectures are based on MCs [158], and PNs [159], [160] but they do not provide a completely automated process. Two of the most commonly used formal verification methods are theorem proving and model checking that we have introduced in Sub-section 2.3.3. Both have been successfully used for the verification of correctness of a broad range of hardware systems. Model-checking tools such as PRISM [161] or UPPAAL-SMC [162] that support formalisms like CTMC or *Probabilistic Timed Automata* (PTA) have been used with success for reliability analysis. Lanfang et al. [163] proposed an approach for the formal verification of TMR FT systems using CSP. By specifying the property of a faultless module using a CSP process, they proved that TMR could still satisfy the property

in spite of hardware errors. The verification process was automated employing the model checking tool FDR2. Hartmanns [164] proposed MODEST, a language based on MCs and PTA that allowed a single model to be analyzed with a range of existing back-end tools like Monte-Carlo simulation or PRISM. With the above formalisms, design space formalization is not possible as they lack generic modeling features and cannot express parametrized systems.

The *Model-Based Safety Analysis* (MBSA) approach of Lisagor et al. [165] addressed these issues by adopting hierarchical modeling, failure modes, and failure propagation as key concepts. Delange and Feiler [166] proposed a safety extension for AADL adding failure mode propagation rules to the design. To model failure mode propagation conditions, Kabiret al. [167] extended a Simulink model with Boolean formulas. In the above mentioned tool xSAP, Bittner et al. [156] annotated a reference functional model with timed failure propagation information.

Since non-functional factors also need to be modeled in order to conduct safety assessment, instead of extending a functional model with safety information, another MBSA group of works proposed dedicated languages for safety modeling. Altarica formalism [168] proposed a hierarchical modeling approach based on components and data-flow, with a semantics based on *Stochastic Guarded Transition Systems* (SGTS). The *System Structure Modeling Language* (S2ML) framework [169] is a prototype-oriented system architecture language that borrowed concepts from *Object-Oriented Programming* (OOP) in order to model the structure of systems.

Nuzzo et al. [170] proposed a formalism based on the failure probabilities of the components, which contribute to the system failure probability based on their degree of redundancy. They implemented the proposed method in the ArchEx framework, based on a high-level pattern-based specification language and MILP-based architecture selection algorithms. They also proposed some optimization schemes to decrease the problem complexity, however they observed that MILP solvers could incur large run-times.

Buyse et al. [171] presented Alpacas, a language for architecture modeling and analysis using SGTS as underlying formalism. It extends the state of the art in model-based safety assessment by offering generic programming features such as encapsulation, generic parameters, and polymorphism, which can allow for supporting design space exploration. Alpacas is domain-specific, anyway similar concepts apply in other domains too.

Mathematical modeling makes FM an accurate and rigorous analysis method

compared to the traditional analytical and simulation-based analysis. However, these benefits are achieved at the cost of heavy computational requirements.

3.3 High Level Synthesis Optimization

In this work, we address the problem of finding an optimized FT configuration of a given system wrt. multiple design objectives. As in most practical cases, the problem under study has a finite number of alternative solutions: thus, it can be formulated as COP. COPs are concerned with the efficient allocation of limited resources to meet desired objectives. They are discrete problems with a set of discrete resources to allocate, and a discrete set of solutions. Constraints on the resources reduce the total set of possible solutions, however, for most problems there is still a great number of feasible solutions. COP are generally extremely challenging computationally. Typically they are NP-hard, and thus cannot be solved exactly in polynomial time (unless $P = NP$). From a theoretical perspective it is difficult to get much traction on these problems. These problems are however a reality and are in fact ubiquitous in our society. Many approaches have been developed to tackle these kinds of problems, and many tools have been designed to aid in the process. However, it is often difficult to predict what will and what will not work. It is unlikely that a single approach will be effective on all problems, or even on all instances of a single problem. In recent years there has been a trend toward automating synthesis at higher and higher levels of the design hierarchy. There are a number of reasons for this:

- Shorter design cycle. If more of the design process is automated, a company can get a design out the door faster, and thus have a better chance of hitting the market window for that design. Furthermore, since much of the cost of ES is in design development, automating most of that process can lower the cost significantly.
- Fewer Errors. If synthesis process can be verified to be correct there is a greater assurance that the final design will correspond to the initial specification. This will mean fewer errors and less debugging time.
- The ability to search the design space. A good synthesis system can produce several designs for the same specification in a reasonable amount

of time. This allows the developer to explore different trade-offs between cost, speed, power and so on, or to take an existing design and produce a functionally equivalent one that is faster or less expensive.

- The design process is self-documenting. An automated system can keep track of what design decisions were made and why, and what the effect of those decisions was.
- Availability of technology to more people. As more design expertise is moved into the synthesis system, it becomes easier for a non-expert to produce a system that meets a given set of specifications.

A review of the literature revealed that problems similar to the one considered, basically a *constraint-based synthesis*, can exist in different application areas. Therefore, we attempt to categorize the commonly found DSE problems based on structure and intent of exploration rather than the application area. Existing DSE case studies found in literature are variants of these classes, or a combination of them.

3.3.1 Configuration or System Assembly Problems

The configuration or *System Assembly Problem* (SAP) consists in selecting and assembling the available components such that all of the system and component requirements are satisfied. Many component-based [172] and platform-based [173] approaches targeting ES have been developed both in academia and in industry. Their life cycle is organized in several phases. Availability of existing components should be considered in requirements specification. During the design phase a component selection strategy has to be defined as this decision plays a crucial role in the architecture design and the resulting quality of the system. Much less time is spent in development, while testing occurs throughout the process to know the effectiveness of the assembled components and check the proper integration of component into the system. Maintenance usually only involves replacement of obsolete component with the new component. For instance, large industrial systems can be assembled by integrating *Commercial Off-The-Shelf* (COTS) components, bringing higher reliability, lower development costs, and shorter development cycles. In general, SAPs require more time during analysis and design phases (see Figure 3.1). Ascia et al. [174] proposed a methodology based on evolutionary techniques for exploration of the range of possible configurations of a parameterized system.

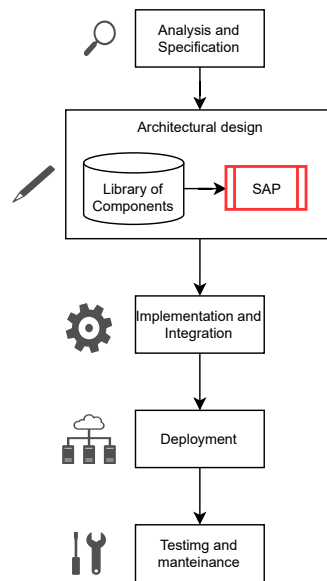


Figure 3.1: System assembly problem as part of the system development life cycle

Initially a population of random configurations is generated. Each configuration is mapped onto the platform and the specific application is executed. The information collected is then used to estimate the variables to be optimized and a fitness function is evaluated. Neema et al. [175] proposed a tool suite for component-based model synthesis named DESERT. It can be interfaced to existing modeling and analysis environments and can be inserted in various, domain specific design flows. The modeling component of DESERT supports the modeling of design spaces and the automated search for designs that meet structural requirements: given a set of models for subcomponents, DESERT composes a system model automatically such that a set of design constraints are satisfied. DESERT is the key tool for DSE included in the toolchain named OpenMETA [176], specifically developed to facilitate the design process for CPSs. Grunske [128] proposed to use evolutionary algorithms and multi-objective optimization strategies to automate the identification of design alternatives that keep cost as low as possible, improve reliability and reduce the weight of the system (this limits the number of redundant components). Manolios [177] presented a fully automated framework named CoBaSA to solve SAP using verification technology that takes advantage of Boolean satisfiability methods. The framework allows the designer to express architectural constraints between components and automatically synthesize architectural models that satisfy these constraints. The way the framework works

Table 3.1: Configuration (or system assembly) problems

Architecture model	Optimization goal(s)	Approach	Tool	Year	Paper
Mathematical	Reliability	Dynamic programming	Not Specified	1968	[178]
Mathematical	Reliability	Dynamic programming	Not Specified	1981	[179]
Application-Specific	Area, Power, Performance	EA	Galib	2002	[174]
UML	Cost	OBDD	DESERT	2003	[175]
Application-Specific	Cost, Reliability, Weight	EA	Not Specified	2006	[128]
Application-Specific	Processor Load.	SAT, PBSAT, ILP	CoBaSA	2010	[177]
Application-Specific	Execution time, Power	Meta-heuristics	C++ language	2011	[180]
Custom	Performance, Power, Cost, Size	GA	SysML	2012	[181]
Application-Specific	Performance, Dimensions, Weight, Cost	OBDD	OpenMETA	2014	[176]
Application-Specific	Circuit parameters	RL	AutoCkt	2020	[182]
Application-Specific	Bandwidth, Noise, Power	GA + NN + MILP	DISPATCH	2020	[183]

is by translating the constraints to a SAT, *Pseudo-Boolean Satisfiability* (PB-SAT), or ILP problem, which can be handled by any of the existing SAT or ILP solvers. The framework can also be used in optimization mode, looking for a solution that is optimal with regards to an objective function. Soto et al. [180], studying the impact on execution time and on power consumption of dynamic memory allocation in embedded processors, proposed two iterative meta-heuristics aiming at determining which data structure should be stored in cache memory at each time interval in order to minimize reallocation and conflict costs. Van Huong et al. [181] presented a new approach to design and optimize ES in the design phase based on Pareto multi-objective optimization. They defined a *Domain Specific Language* (DSL) and developed a framework which is used to design the architecture model of ES. And integrated the code generation technology called Text Template Transformation Toolkit to this framework to automatically generate parameters from architectural model. Then, used multi-objective optimization to select the best trade-off configuration of the architecture based on Pareto principle and GA. Settaluri et al. [182] proposed a *Machine Learning* (ML) optimization framework trained using deep *Reinforcement Learning* (RL) combined with transfer learning over a sparse design space to synthesize analog circuits. RL is a branch of ML in which a model (aka policy) learns to take actions in an environment to maximize a given reward function. Terway et al. [183] addressed the design of CPSs as a two-step multi-objective optimization problem: first, they applied a genetic algorithm to search over a discrete set of choices for component selection, thus yielding a coarse design. In the second step, they used an inverse design to search over a continuous space to fine-tune the component values and met the diverse set of system requirements. They used a NN as a surrogate function for the inverse design of the system. The NN, converted into a MILP, is used for active learning to sample component values efficiently in a continuous search space.

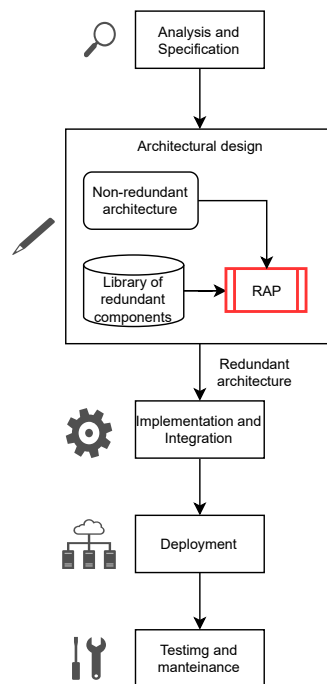


Figure 3.2: Redundancy allocation problem as part of the system development lifecycle

3.3.2 Redundancy Allocation Problems

The goal of *Redundancy Allocation Problem* (RAP) consists in finding an optimized FT configuration of a given system wrt. one or multiple design objectives. Although non-functional requirements like performance or reliability are highly dependant on the implementation and deployment phases, discovering and evaluating redundant design alternatives can be captured at design phase. The introduction of this abstract information at the early stage of development (see Figure 3.2) is useful for discarding unsuitable design points and thus reducing the design space of feasible candidates.

Fyffe et al. [178] used a dynamic programming approach to solve a RAP for a system with fourteen subsystems and constraints on cost and weight. For each subsystem, there were three or four component choices, each with different values of reliability, cost, and weight. To accommodate multiple constraints, they used a Lagrangian multiplier within the objective function. Instead of Lagrangian multiplier,

Nakagawa and Miyazaki [179] used a surrogate constraints approach that should delimit the general shape of the solution more quickly. They demonstrated their algorithm by solving thirty-three variations of the Fyffe problem. These formulations provided a selection of components, but the search space

was restricted to consider only solutions consisting of the same component type used in parallel. These formulations provided a selection of components, but the search space was restricted to consider only solutions consisting of the same component type used in parallel.

One of the first use of a meta-heuristic for reliability optimization was proposed by Coit et al. [184]. In order to identify the choice of components and design configuration in a S-P system, they addressed the RAP using a combined NN and GA approach. The GA searches for the minimum cost solution by selecting the appropriate components, given a minimum system reliability constraint. A NN estimates the system reliability value during search. Components could be mixed flexibly within subsystems: this feature enabled the GA to find solutions with higher reliability than the exact method above: this feature enabled the GA to find solutions with higher reliability in twenty-seven out of thirty-three problems than the exact methods above.

In order to identify the choice of components and design configuration in a S-P system, Coit et al. [184] Kulturel-Konak et al. [185] proposed TS as the DSE strategy to address the same problem. Liang et al. [186] used an ACO metaheuristic method. Jhumka et al. [187] introduced a GA to solve the MOOP taking into account reliability together with performance and costs. The system hardening was performed as a second step after the synthesis of the nominal system; in particular, replica tasks were scheduled in the free time slots of the processing units to avoid performance degradations. Some authors [188, 189, 190] adopted a probabilistic fault model in which, starting from a behavioral description in form of a *Data Flow Graph* (DFG) and a resource graph for modeling all architectural alternatives, each processing unit was characterized in terms of failure probability; then, task replication and binding to processing units was performed within a multi-objective DSE to concurrently optimize reliability, area, and performance. Izosimov et al. [191] proposed a DSE approach combining different techniques (task replication, transparent re-execution, and optionally checkpointing) to optimize system performance. They used a *Directed Acyclic Graph* (DAG) to model an application running on a set of computation nodes connected via a bus. The author adopted a complete FT requirement to deal with a specified number of transient faults and adopted a target architecture consisting of units with fault detection mechanisms. In their framework named ArcheOpterix, the authors [192] considered reliability, cost, and response time as optimization criteria and employed a multi-objective ACO algorithm. Sheikhalishahi et

Table 3.2: Redundancy allocation problems

Architecture model	Optimization goal(s)	Approach	Tool	Year	Paper
S-P of k-o-o-n	Reliability	NN + GA	Not specified	1996	[184]
S-P of k-o-o-n	Reliability, Cost, Wheight	TS	Not specified	2003	[185]
S-P of k-o-o-n	Reliability, Cost, Wheight	ACO	C++	2004	[186]
Task Graph	Reliability, Performance, Cost	GA	N.A.	2005	[187]
DFG	Reliability, Area, Performance	EA	PISA, LpSolv, Forte's Cynthesizer	2007/8	[188], [189], [190]
DAG	Cost	Task replica	N.A.	2009	[191]
DTMC	Reliability, Cost, Time	ACO	ArcheOpterix	2009	[192]
Graph-based	Reliability, Exec.time, Energy	Scenario-based	SAFE	2012	[200]
S-P of k-o-o-n	Reliability, Cost, Volume, Wheight	PSO	Matlab	2013	[193]
Application-Specific	Reliability	SMT	KCR (Z3-based)	2014/17	[194], [195]
S-P of k-o-o-n	Reliability	EA + MC	Matlab	2018	[196]
Logical formulae	Safety, Energy, Cost, etc.	SMT	N.A.	2018	[197], [198]
Multiple S-P	Reliability	MA	N.A.	2020	[199]

al. [193] applied a hybrid between a GA and PSO to a MOOP minimizing cost, system volume, and weight, while maximizing system reliability. Delmas et al. [194, 195] proposed an automatic SMT-based DSE method to harden an initial architecture for a given safety objective. Given the nominal functional architecture, constraint solving was used to select automatically a subset of system components to update and appropriate safety patterns to apply to meet safety requirements. Ardakan and RezvanIn [196] addressed the RAP by using a multi-objective EA algorithm (NSGA-II) combined with a Markov-based approach. Results demonstrated that their algorithm leads to high reliability, but a standby strategy is developed for the problem. Terzimehic et al. [197], [198] presented a method of SMT-based automated deployment of industrial automation systems. In the context of the control applications, they encoded the problem into an SMT form and validated it, although with their approach they encountered scalability issues. Zaretalab et al. [199] proposed a mathematical model for optimizing multiple redundancy-reliability systems known as mega-systems. They used a parameter-tuned *Memetic Algorithm* (MA) to solve the problem. According to their results, the computational time of MA was higher than commonly used GA, but performance in terms of the quality of the obtained solutions was better.

3.3.3 Selection Problems

Given a set of objects, the goal of a *SeLection Problem* (SLP) is to find a subset of it, such that a set of constraints are satisfied and a cost function (for example cost, memory consumption etc.) is optimized. Selection problems have been commonly found in SPL engineering where the set of features are organized in a tree-like structure called the *Feature Model* [201]. The solution of a feature SLP is a subset of the features (configuration) that satisfy all the

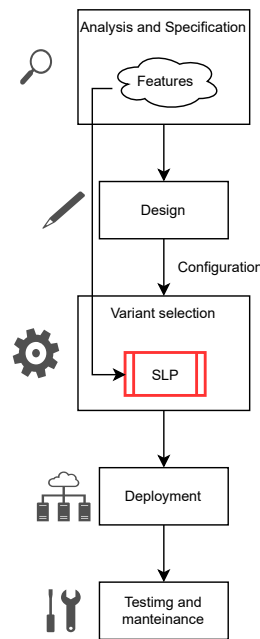


Figure 3.3: Selection problem as part of the system development lifecycle

constraints and optimize the total resources required by the product. Deciding which features should be supported by the product line is performed at requirement-level. The semantics of features can be captured by a structural or behavioral model. This idea leads to the concept of feature-based model templates [202]. A selection is usually performed over a given configuration and it is the set of products of the SPL that are valid for the given configuration (see Figure 3.3). For instance, White et al. [203] presented a tool called Scatter whose input is (1) the requirements of *Product-Line Architectures* (PLA) construction and (2) the resources available on a discovered mobile device and whose output is the optimal variant that can be deployed to the device. Seidl et al. [204] leveraged *Delta-Oriented Programming* (DOP) - a programming language approach particularly designed for implementing SPLs by defining a core module and delta modules. [205] - and proposed DeltaEcore, a framework to automatically derive delta languages to express the delta operations to the common core of the product line. Then, products were configured based on a hyper feature model, which provided the option to specify revisions of individual features. Pietsch et al. [206] presented a novel approach to delta modeling, an implementation technology able to generate models as instances of an SPL, and a supporting tool suite. They refined the abstract notion of a delta to be a consistency-preserving edit script generated by comparing two models,

Table 3.3: Selection problems

Architecture model	Optimization goal(s)	Approach	Tool	Year	Paper
Feature Models	Availability, Reliability, Cost, etc.	Commercial CSP solver	OPL	2005	[201]
PLA	Cost	CSP + branch and bound	Choco Solver	2007	[203]
Feature Models	Performance	Delta modeling	DeltaEcore	2014	[204]
Feature Models	Performance	Delta modeling	SiPL	2015	[206]
Feature Models	Performance	Test-based	FLAME	2017	[207]
Feature Models	Performance	SAT	FeatureIDE	2017	[208]
Feature Models	Cost, LOC, N.of faults, N.of installations	GP	Prototype tool based on NSGAI	2017	[209]
Feature Models	Modifications availability	Model-driven	SuperMod	2017	[210]

allowing them to detect conflicts and further relations between deltas. Duran et al. [207] presented a proposal to model and reason on SPL using *Constraint Programming* (CP), taking into account functional and extra-functional features: the FLAME framework. It can be used to formally specify not only feature models, but other VMLs as well. Schroter [208] proposed the concept of a feature model interface that only consists of a subset of features and hides all other features and dependencies. Based on a formalization of interfaces, he proved compositionality properties: the key towards better scalability. Kitefew [209] proposed a novel encoding of candidate solutions, based on grammar representation of feature models, which ensured that relations imposed in the feature model were respected by the candidate solutions. Specifically, their approach exploited the potential of *Genetic Programming* (GP) to evolve structurally valid individuals derived from a grammar, hence they first transformed the feature model to an equivalent grammar, then they applied *Grammar Guided GP* (gppp) to evolve candidate solutions towards ultimately finding the optimal set of product configurations. Each candidate solution (product configuration) is derived from the grammar, and hence by definition respects the constraints imposed by the feature model. Schwagerl et al. [210] proposed a framework for managing evolving model-driven SPL for the domain of software configuration management. It provided higher-level abstractions in the form of models, which automatized great part of version management. Their tool utilized feature models in order to define logical variants and constraints. The objective was to make the performed modifications available for a larger set of related variants.

3.3.4 Placement Problems

Given a set of two-dimensional objects and a plane, the *Placement Problem* (PLP) is to assign a position to each object, such that all objects lie within the given boundary of the plane and the objects do not overlap. In a variant of the problem, the plane is not given and the objective is to minimize the area of the

enclosing box containing all objects. PLPs have been found most often in *Very Large Scale Integration* (VLSI) design. Although there are many variations in PLP formulations for different design styles and with different objectives, the underlying issues are the same: find the exact location of the cells to minimize area and wire length. After the design has been successfully synthesized into a gate-level netlist, and a floorplan with pre-placed blocks has been designed, we can move into detailed placement of the standard cells of physical design implementation. Jiang et al. [211] presented example analytical techniques employed in the leading academic placer, NTUplace3, to face the VLSI placement problem where the designer has to handle large-scale designs with millions of objects, heterogeneous objects with very different sizes, and various complex placement constraints such as preplaced blocks and chip density. Saraswat et al. [212] proposed a finite domain constraint-based placement tool that makes placement decisions cognizant of communication delays. They exploited the deterministic properties of the communication network and formulate them as finite domain constraints to translate the placement problem into a CSP, which can be solved using a constraint solver. Chen et al. [213] presented an *Adaptive Hybrid Genetic Algorithm* (AHGA) for VLSI standard cell placement problem, which used some adaptive strategies to reduce the runtime on the premise of obtaining the same high-quality placement results of analytical algorithms. More recently, Goldie and Mirhoseini [214] proposed RL with policy gradient optimization as a solution to the placement problem.

3.3.5 Routing Problems

Given a graph $G = (V; E)$, where V is a set of nodes, E is a set of non-negative edges, and a set of terminals $T \subseteq V$, the routing problem is to find the a tree connecting the nodes in T , such that the tree is optimal with respect to an objective. These problems are commonly found in wire routing in VLSI [215] or network design [216]. Routing problems are often found in VLSI design in combination with placement problems (see Figure 3.4): given a placed netlist, a signal routing solution determines the necessary wiring to connect cells while meeting design rule constraints and routing resource capacities. As a consequence, the non-functional properties (e.g. latency, energy consumption, area requirements) of the resulting system implementation may vary considerably depending on high-level decisions that have been made. Hence, a DSE is

imperative to find optimal solutions. In the literature both exact and approximation have been proposed. Chang et al. [217] used dynamic programming to find optimal paths that satisfy the minimum area rules and an end-of-line spacing constraint. Kahng et al. [218] solved the resulting instance of the detailed routing problem using an ILP-solver. A different approach was used by Zhang and Chu [219] that proposed a heuristic to solve a given routing problem. Liu et al. [220] employed a discrete PSO algorithm. In general, exact methods are often replaced in favor of heuristic approaches as the complexity of systems increases. On the other hand, approximation methods usually create the initial population with a randomized process, and execute the search on the basis of previously found solutions, not systematically. This could lead to run into saturation. For this reason Neubauer et al. [221] proposed to encode the problem symbolically, leveraging advances of constraint solving technologies, specifically ASP. According to the authors, using ASP, rather than other symbolic techniques like SAT, fastened the communication synthesis because reachability was expressed naturally.

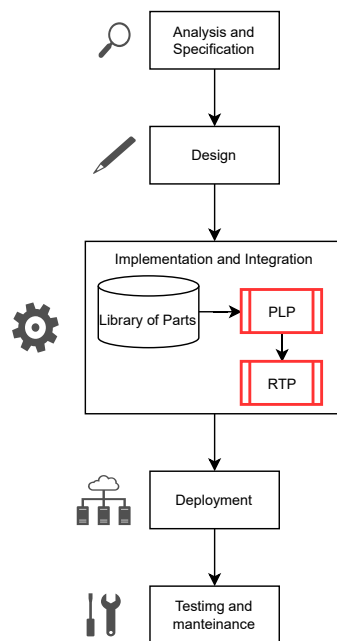


Figure 3.4: Placing and routing problems as part of the system development lifecycle

Table 3.4: Placement and routing problems

<i>Architecture model</i>	<i>Optimization goal(s)</i>	<i>Approach</i>	<i>Tool</i>	<i>Year</i>	<i>Paper</i>
Hypergraph	Preplaced blocks, Chip density, Routability timing, Power.	Gradient method with dynamic step-size	NTUplace3	2007	[211]
Operation Dependence Graph (ODG)	None	Finite domain CSP solving	Mozart/Oz	2008	[212]
Graph-based	Power consumption	MILP	GAMS	2008	[216]
Data Flow Graph (DFG)	Area, Communication delay, Schedule length, Routing resources	Finite domain CSP solving	Mozart/Oz	2010	[215]
Hypergraph	Routability, wirelength	Hierarchy grouping and clustering	NTUplace4h	2014	[222]
Hypergraph	-	Finite domain CSP solving	ePlace	2014	[223]
Graph based	runtime	AHGA	C language	2016	[213]
Hypergraph	Routability, wirelength, timing, power	Constraint-oriented local smoothing and dynamic step size adaptation	RePlace	2016	[224]
Hypergraph	-	Deep learning	Dreamplace	2020	[225]

3.3.6 Deployment Optimization Problems

In general, a deployment refers to the assignment of hardware resources, e.g. CPU time, memory or I/O, to software components. This is a hardware to software deployment, which faces the problem of which software component to deploy on which execution unit. There is also the case of hardware to hardware deployment, e.g. which sensor or actuator shall be connected to which device. The deployment optimization problem is about how to configure and deploy applications correctly while at the same time optimizing a few criteria. Kruchten [226] pioneered the correspondences between architecture views early in 1995. Clements et al. [227] modernized Kruchten’s approach to define deployment viewtypes in order to allocate software modules into runtime. Kramer et Magee [228] motivated automatic deployment for autonomous systems while [229] focused on dynamic re-conguration using AI planning. Carlson et al. [230] presented an approach to combine model-driven and component-based software engineering to perform incremental deployment of model-concepts to runnable entities. Other authors focused on the impact of quality factors for deployment, such as Bushehrian [231] and White [232] that used performance to compute the nearest optimal deployment using simulation and EA respectively. Petricic et al. [233] studied deployment mechanisms in a context of runtime reconfiguration. In such a context a distinction is usually made between functional and structural changes. Functional changes include the addition of new/improved code to a running system in order to modify/upgrade its functionality, whereas structural changes refer to re-configurations that change the relationship between different components of the system, or replicate portions of the application for execution on a different machine. The focus of their research is on functional changes. Their main objective was to develop a dynamic deployment mechanism for ES that is resource efficient and ensures predictability of system behavior by introducing performance attributes

verification of deployed components. They performed component verification during deployment to be able to predict system's behavior after reconfiguration. The authors were particularly interested in verification of non-functional properties like CPU share, memory, energy, bus bandwidth, which are critical for ES. Recent approaches showed examples of automatic deployment using SAT solvers to optimize the best deployment configuration. Kugele et al. [234] presented and compared three different approaches: a MOEA, a SMT-based, and an ILP-based approaches, to tackle the deployment problem. They also used a combination of the first two. They concluded that deciding which approach is best suited depends on the size of the model, constraints, and objectives. Zverlov [235] presented an approach to find an optimized platform, deployment, and schedule wrt. to safety, cost, performance, and resource consumption, using a meta-search on top of a SMT solver, which manipulates constraints. Abraham et al.[236] presented an automatic configuration generator tool named Zephyrus2 that the designer can use to specify requirements in the form of constraints and optimal system configurations and deployment at minimal cost. The tool solves the resulting MOOPs by optimizing the first objective function value and then optimizing the other objective function values sequentially following their order after substituting the previously determined optimal values. According to the authors, minimizing the first objective (e.g., the cost) has a significant impact on the performance when reducing the second objective (e.g., the number of components). However, this solution has the drawback that the designer needs to restart the solver. Terzimehic [198] and Gruner [197] presented a method of SMT-based automated deployment of industrial automation systems. In the context of the control applications, and more specifically with reference to IEC61499-based systems, they encoded the deployment problem into an SMT form and validate it, although with their approach they encountered scalability issues.

3.3.7 Resource allocation Problems

Given a set of objects and a set of resources, the goal of a resource allocation problem is to allocate resources to the objects, such that all constraints are satisfied. After a system has been implemented, i.e. the software components have been generated, part of the deployment activity deals with mapping the software components onto hardware nodes (see Figure 3.5). This is one of the crucial aspects that influence reliability of ES. If the hardware architecture is

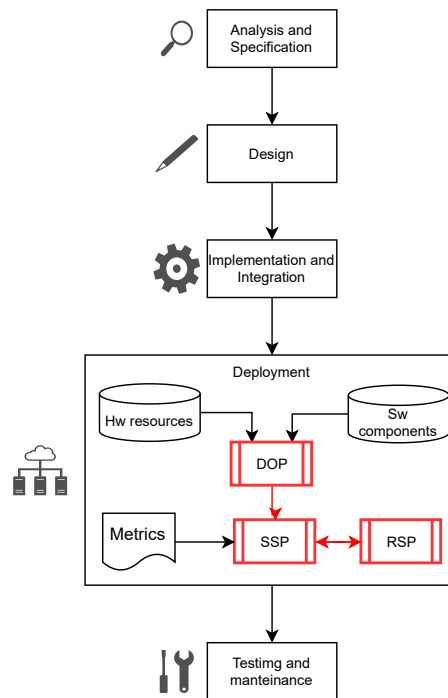


Figure 3.5: Mapping and scheduling problem

designed prior to the customized software architecture, which is often the case in product-line manufacturing (e.g. in the automotive domain), the system architect needs to resolve a nontrivial task of finding a (near-)optimal deployment balancing the reliabilities of individual services implemented on the software level. In many domains of ES, task allocation is typically addressed as a sub-problem of scheduling activity. In any case, determining feasible allocation schemes requires knowledge of both functional and non-functional properties of the system. And the candidate architectures must be synthesized, before analyzing their performances for trade-offs. A recent trend is that of dynamic allocation of resources. The evolution of submicron technology has dramatically increased IC density, enabling the development of *Multi-Processor Systems on a Chip* (MPSoC)s that support a variety of multimedia and networking applications that present a dynamic task workload. This implies a varying number of tasks running at any given moment, possibly exceeding the available hardware resources. Hence, it is necessary to control system resource use, including dynamic management of task-loading actions, which can drastically influence system performance. If we use the moment at which a task is defined as a classification criterion, task-mapping approaches can be either static or dynamic. Static mapping defines task placement at design time; dynamic mapping defines task placement at runtime. The topic of run-time

(or on-the-fly) mapping has received substantial research attention in recent years [237, 238]. In these methods, the assignment of newly arriving tasks to the system resources is done by means of heuristics. Traditional design-time mapping solutions usually produce higher quality mappings as they allow for exploring a larger design space for the underlying architecture. But as these algorithms typically involve slow computational methods [239] such as ILP, they cannot be used during run time. Other research directions have been investigated that propose a mixed design-time and run-time approach that reuses the analysis results obtained at design time to accelerate and improve run-time mapping [240, 241, 242]. More in general, different resource allocation problems found in the literature can be classified into two subcategories based on the allocation relation:

- A *Relational* Allocation Problem has a many-to-many relation between components and resources, such that a resource can be allocated to many components and a component requires more than one resource. Frisch et al. [243] proposed the *Synchronous Optical NETWORK* (SONET) design: an example of a relational resource allocation problem. The goal of the SONET problem is to allocate a set of nodes to one or more optical network rings. Weichslgartner et al. [244] proposed a hybrid approach that considers constrained shared communication and computation resources. It uses a backtrack algorithm and targets specific goals by minimizing the communication distance in the mappings. Kirov et al. [62] introduced ArchEx 2.0, an optimization-based framework for architecture exploration where these architectures are composed of components that use flow-based communication. The task is to find the correct number of components, select an implementation for each component from a library, and connect them together in order to minimize an objective function while guaranteeing that system requirements are satisfied. Goens et al. [245] proposed TETRiS for static mappings for heterogeneous system-on-chip architectures, which allowed them to achieve energy efficiency by focusing on remapping and migration of task in dynamic multi-application scenarios.
- A *Functional* Allocation Problem is an allocation problem where every component requires exactly one resource. Each component can require only one resource, but more nodes can require the same resource. The

Table 3.5: Resource allocation problems

<i>Architecture model</i>	<i>Optimization goal(s)</i>	<i>Approach</i>	<i>Tool</i>	<i>Year</i>	<i>Paper</i>
Sets and functions, matrix-based	None	CSP solving	CGrass	2002	[243]
Kahn Process Network (KPN)	Processing time, Power Consumption, Cost	EA	PISA	2006	[246]
Graph based	None	Ad hoc algorithm	Edipe	2008	[247]
Directed graph	Energy consumption	Meta-heuristic (Opt4J-based)	DAARM	2014	[244]
Directed graph	Cost	MILP	Archex	2017	[62]
Graph based	Resource usage, CPU time, Execution time	Heuristics	TETRiS	2017	[245]
Analytical	Data rate	MINLP + pre-emptive cut generation	N.A.	2017	[248]

mapping problem faced by Erbas et al. [246] is a case of functional resource allocation problem. Another example is CPRTA [247] (for "Constraint Programming for solving Real-Time Allocation"): an original approach based on constraint programming to solve a static allocation problem of hard real-time tasks. This problem consists in assigning periodic tasks to distributed processors in the context of fixed priority preemptive scheduling. CPRTA is built on dynamic constraint programming together with a learning method to find a feasible processor allocation under constraints. CPRTA exhibits very interesting properties. It is complete (if a problem has no solution, the algorithm is able to prove it); it is non-parametric (it does not require specific tuning), thus allowing a large diversity of models to be easily considered. In addition, thanks to its capacity to explain failures, it offers perspectives for guiding the architectural design process. Letchford et al. [248] considered a resource allocation problem arising in mobile wireless communications. The goal is to allocate the available channels and power in a so-called OFDMA (orthogonal Frequency-Division Multiple Access) system, in order to maximize the transmission rate, subject to quality of service constraints. A novel ingredient of their algorithm, which turned out to be crucial, is what they call pre-emptive cut generation: the generation of cutting planes that are not violated in the current iteration, but are likely to be violated in subsequent iterations.

3.3.8 Scheduling and Sequencing Problems

Scheduling and sequencing problems are a common category of DSE problems often found in ES [249, 250]. Scheduling deals with defining which activities are to be performed at a particular time. Sequencing concerns the ordering in which the activities have to be performed. In general, these problems are characterized by assigning start times to a series of tasks that have to be performed

Table 3.6: Scheduling and Sequencing problems

<i>Architecture model</i>	<i>Optimization goal(s)</i>	<i>Approach</i>	<i>Tool</i>	<i>Year</i>	<i>Paper</i>
Hypergraph	Memory usage	Genetic algorithm	Lycos	1999	[249]
Task graph	Cost	Heuristic	CHIP	1999	[250]
Data Flow Graph (DFG)	Time	Satisfiability	JaCoP	2003	[252]
ESMoL	None	Constraint logic programming	ESched	2009	[253]

by some deadline with the possibility of precedence constraints between them. The system architecture can be validated (through simulation or verification) to evaluate different scheduling approaches (e.g. in terms of timing) as part of system design space exploration. However, several variants exist, depending on the constraints imposed due to the properties of the tasks and resources. Broadly, there are two kinds of resource and tasks [251]:

- A *disjunctive resource* can execute at most one task at each point in time. The tasks which require this resource can execute only when no other task is executing on the resource.
- A *cumulative resource* can execute several activities in parallel, provided the resource requirement of the executing tasks does not exceed the resource capacity at any point of time.
- A *non-preemptive task* must execute without interruption from start to end.
- A *preemptive task* can be interrupted by other task depending on some priority.

Most real problems consist of a combination of cumulative and disjunctive resources as well as both interruptible and non-interruptible activities. For example, Kuchcinski et al. [252] presented a constraint solver engine named JaCoP (Java Constraint Programming) and a related framework that make it possible to model different resource assignment and scheduling problems, and handle them uniformly. They describe a new method that addresses assignment of resources for operations and tasks as well as their static, off-line scheduling. Different heterogeneous constraints are considered. These constraints can be grouped into two classes: problem-specific constraints and design-oriented constraints. They are uniformly modeled by finite domain (FD) constraints and solved using related CP techniques. Porter et al. [253] presented a prototype scheduling tool (ESched) which calculates cyclic schedules for time-triggered networks. ESched supports the model-based workflow

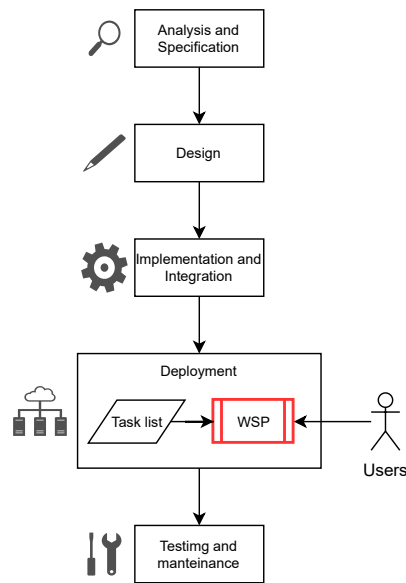


Figure 3.6: Workflow Satisfiability problem

of the ESMoL modeling language and tool suite. Using ESMoL, designers can rapidly iterate through simulating a control design, capturing platform effects in models, generating a schedule (if feasible), and re-simulating the control design subject to the platform model and the computed schedule.

3.3.9 Workflow Satisfiability Problems

A workflow is a collection of steps that must be executed in some specific order to achieve an objective. The execution of each step in a workflow instance can be triggered by a human user, or a software agent acting under the control of a human user (see Figure 3.6). Workflows equipped with an authorization policy and constraints may be called “security-sensitive” [254]. Authorization policies and constraints are fundamental for security, but could lead to situations where a workflow instance cannot be completed because no task can be executed without violating either the authorization policy or the constraints. The *Workflow Satisfiability Problem* (WSP) consists of checking if there exists an assignment of users to tasks such that a security-sensitive workflow successfully terminates while satisfying all authorization constraints. The WSP has many variants and different possible solutions depending on how security-sensitive workflows are specified and on the relationship between workflow model and authorization [255]. Firstly, a workflow specification can

Table 3.7: Workflow satisfiability problems

<i>Architecture model</i>	<i>Optimization goal(s)</i>	<i>Approach</i>	<i>Tool</i>	<i>Year</i>	<i>Paper</i>
Role-and-relation-based access control model (R^2BAC)	None	SAT		2010	[259]
Markov Decision Process (MDP)	Users	SAT		2014	[260]
Directed acyclic graph (DAG)	Users	SAT		2015	[261]
Directed acyclic graph (DAG)	None	Fixed-parameter algorithm		2017	[263]
Petri net	Costs	BMC, OMT		2018	[262]

have different perspectives [256]. The control-flow perspective describes the execution order of the tasks (e.g., sequential, parallel, or alternative execution). The data-flow perspective specifies the data objects used by the tasks. The resource (or authorization) perspective specifies the policies for the task execution. Furthermore, these three dimensions are interconnected, as each one of them influences the others. In addition, different WSP can be formulated by considering order or unordered tasks [257], and by checking the satisfiability at design- or run-time [258]. Wang et al. [259] demonstrated that the WSP is NP-complete even with simple constraints and reduced the problem to SAT. Mace et al. [260] provided quantitative measures indicating a degree of satisfaction and/or resiliency for a given workflow, instead of simply returning an assignment if one exists. This can be helpful with real-life problems where the ideal case is not always reachable. Crampton et al. [261] defined the valued workflow satisfiability problem (valued WSP), whose solution is an assignment of steps to users of minimum cost. Bertolissi et al. [262] solved the Multi-Objective Workflow Satisfiability Problem (MO-WSP) using *Bounded Model Checking* (BMC) and OMT solving.

3.4 Approaches to Automatic Verification

The goal of verification activities is complete coverage and standards conformity, but it is not an easy task. *Formal Verification* (FV) provides methods and techniques to prove mathematically the correctness of a system. Teams use several techniques to assure that no errors exist when the product is delivered. In particular, for critical systems further measures must be taken into account, as these systems deal with human lives. In the following, we provide a review of the most widely used formal verification techniques in the literature.

3.4.1 Automated Theorem Proving

Theorem Proving relies heavily on high order logic and uses mathematical structures to build formulas that correspond to the behavior of the system [264]. The verification process is the evaluation of these formulas. There are different formal logics to describe the theorems. The most common ones are: propositional logic, temporal logic, first-order logic, high-order logic.

The first widely distributed, high-performance theorem prover for first-order logic was *Otter* (acronym for Organized Techniques for Theorem-proving and Effective Research) [265], developed by William McCune at Argonne National Laboratory in Illinois. In their work [266], Owre et al. present PVS, a prototype system for writing specifications and constructing proofs. PVS combines high-order logic with a highly interactive proof checker that supports top-down proof exploration and construction. Mentré et al. [267] combine the B Method [268], a formal approach to develop safety critical ES by refinement techniques, with SMT solvers that automate the proving process. In the work of Wiedijk [269], other examples of Automated Theorem Proving tools can be found.

3.4.2 Symbolic Model Checking

Model checking verifies properties against a model to prove that the design conforms to the specifications. It is very powerful, although its method of explicit state enumeration is highly resource consuming. For large systems with thousands of states, the number of possible states that the model checker can work on is therefore limited. Instead of explicitly enumerating all states, *Symbolic Model Checking* (SMC) works with sets of symbols. It uses BDDs to represent sets of states and to work with them in bulk operations. This way, it is possible to verify systems that are more complex. BDDs represent Boolean functions in a canonical form, where a path can be traced from the root node to any of the leaf nodes so that it can easily determine if a path satisfies the Boolean function or not. However, BDDs are highly dependent on the variable ordering, which directly affects their size. Therefore, it is necessary to apply good strategies and heuristics to optimize them correctly. The result of a symbolic model checker when it fails to verify the input is a counterexample, which traces a path from the initial state to the offender state, or the state that caused the proposition to fail. With this trace, a developer can recreate the failure and find out what caused it. SMC uses temporal

logic to describe the propositions to verify the system’s behavior, together with the model that describes the system itself. Once the formulas are built, the generated BDDs are traversed to reach one of the two possible leaf nodes: “true”, meaning that the property holds, or “false”, meaning that the property does not hold. When a property is false, a counterexample is generated, and the designer can either fix the system architecture or refine the propositions, in the case of a false positive result. The propositions are fed into the tool, which means that the process is automated. An example of tool based on SMC is *NuSMV2*. It was designed to be very robust, easy to modify, and close to the standards required by industry [270]. The input descriptions are done in SMV language, the tool that preceded and served as the base for NuSMV2. It is possible to use modules and processes to describe finite state machines while incorporating requirements in LTL and LTL. NuSMV2’s flow has several steps for transformations and optimizations. Steps performed by NuSMV2 are shown in Figure 3.7. The first three steps, flattening, Boolean encoding, and cone of influence, refine the model M and the properties P_1, \dots, P_n into modules and processes. The result is a synchronous flat model where only the areas inside the cone of influence for each property is considered. This helps to reduce the system complexity and manage the state space size. The resulting model is applied either to BDD- or to SAT-based model checking. For BDD-based verification, the tool builds a BDD representation of the system (step BDD-based Model Construction) and then verifies it (step BDD-based verification). For SAT-based verification, NuSMV2 builds a representation of the model to apply it to the model checker (step BMC). If a counterexample is found, NuSMV2 transforms it to CNF and feeds it to a SAT solver.

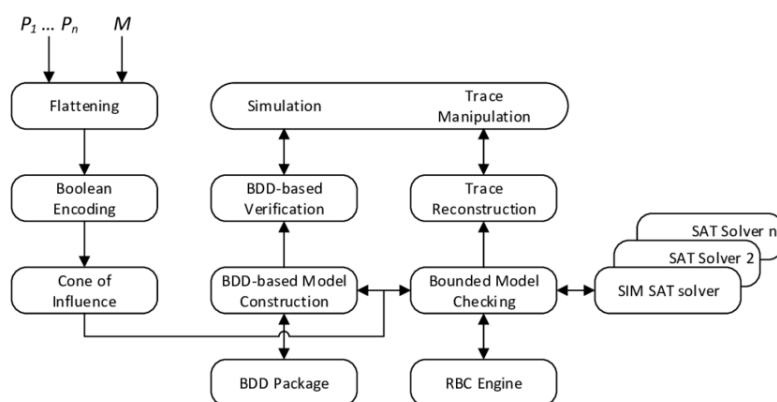


Figure 3.7: Steps performed by NuSMV2 [270].

nuXmv is an evolution of NuSMV. According to [271] capable of dealing with finite- and infinite-state systems. It has been used as the back-end application for many different tools, both academic and industrial. Another tool based on SMC is *Uppaal*. It deals with real-time systems, which can be expressed using the theory of timed automata. Like NuSMV2, Uppaal has its own specification language. However, it uses a modeling language for the specification of the models and a query language for the properties. The modeling language was designed to facilitate the description of finite state machines, where states, clocks and the interactions between them can be expressed. The query language uses a subset of CTL for the state and the path formulas, where the state formulas are properties for individual states and path formulas are properties for traces of the model. As stated by Behrmann et al. [272], Uppaal consists of three parts: the editor, the simulator, and the verifier. In the editor, the user describes the model for the network of timed automata that characterizes the system with its relations of synchronization and update between the states. The simulator can be used to run the system manually and choose the path to follow, or to let the system run at random, or to use a saved trace to check the reachability of a given state. The verifier lets the user add and edit the properties, or “queries” in Uppaal’s interface, for verification and check them.

3.4.3 Bounded Model Checking

BMC is a technique that tries to overcome the problem caused by the state space explosion of SMC [273]. BMC tries to find counterexamples within a bounded length k , which, in turn, generates minimal counterexamples. This way, when verifying a system, paths are walked through up to a length k in order to check its correctness, instead of traversing all states up to the same point. This prevents the state space explosion up to length k . On the other hand, as BMC searches the states up to a given bound, bugs in deeper states tend to remain hidden. For this reason, a proof using BMC can be incomplete. BMC uses propositional decision procedures (SAT) to model the system, which, as BDDs, are also based on Boolean expressions, but do not try to build a canonical data structure. Because of this, SAT can handle propositional satisfiability procedures with thousands of variables. In BMC, SAT and temporal properties are used to verify the correctness of the model. In this way, propositional formulas are generated if and only if a counterexample

exists. That is, if all the to-be-verified properties hold, then a counterexample does not exist.

CBMC is an example of tool based on BMC. As specified by Clarke et al. [274], It aims at model checking of ANSI-C programs. One of this tool's focus is the verification of functional software prototypes of hardware architecture written in ANSI-C. It also provides a graphical interface. In the latest versions, *CBMC* provides support for external SMT solvers.

Another example of BMC-based tool is *EBMC*, which aims at the verification of hardware designs written in Verilog. It translates the Verilog code to ANSI-C. At this stage, an intermediate representation is generated and it is possible to apply different tools to perform the verification process, either SAT or SMT solvers. This intermediate representation also makes this flow compatible with established industrial tools. See [275] for more details.

3.4.4 SMT Model Checking

Theorem Proving and Model Checking are very powerful techniques. However, due to several drawbacks such as high degree of knowledge of the system, high specialization in higher order logic, and low degree of automation due to complicated formulas, new approaches were proposed. One technique that has been rapidly evolving in the last years is SMT, which is the problem of deciding the satisfiability of a first-order formula with respect to some decidable first-order theory [276]. First-order formulas are SAT procedures and deal very well with the satisfiability part. First order theories, on the other hand, work with predicates and help to create decidable logic that is more expressive. Some examples of theories are Linear Arithmetic, Difference Logic, Unit-Two-Variable-Per-Inequality, Bit-Vectors, and Arrays. One important contribution to this technique is the creation of a library to standardize SMT tools, which help to spread it among different areas and apply the tools for different purposes. SMT-based tools use an SAT solver and one or more theory solvers, depending on which theories are considered. The input to the tool is a theory-formula, which generates a Boolean abstraction. This Boolean abstraction feeds the SAT solver, which enumerates clauses in a collection. Each clause is fed to the theory solver. If a clause is theory-satisfiable, the solver returns a positive response; otherwise, it updates the Boolean abstraction. This process is repeated until a complete theory-satisfiable Boolean abstraction is found, meaning it is satisfiable ("SAT"), or if no more updates can be made,

meaning it is not satisfiable (“UNSAT”). Examples of SMT-Based Tools are: *MathSAT5*, *nuXmv*, *ESW-CBMC*.

3.4.5 OMT Model Checking

The rise in efficiency SMT solvers has produced numerous uses for them in many areas of Computer Science and Engineering. SMT solvers allow us to encode the problem as CSP. However, often we are interested in finding not just an arbitrary satisfying assignment, but one that optimizes (minimizes/-maximizes) certain criteria, i.e. that is optimal wrt. some objective functions. SMT problems of this kind are referred to OMT. Thus, our problem can be encoded as COP. OMT is an extension of SMT wthat allows for finding models that make a given objective optimum through a combination of SMT and optimization procedures [277], [278], [279], [280], [98].

3.4.6 Equivalence Checking

Equivalence checking is widely used in the industry. It aims to prove if two implementations on different abstraction levels are functionally equivalent [281]. Nowadays, implementation of prototypes on higher levels of abstraction is very common as a starting point to begin the development and analysis of new architectures, as they are easier to build and debug. Furthermore, as the refinement of these prototypes gets closer to the lower levels they become an important reference model to the implementation. Equivalence checking tools can be implemented using one or a combination of several techniques; the most prominent are BDD-, SAT-, structural- and signature-based. In BDD-based tools, each implementation generates a BDD and are then compared to each other For SAT-based tools, the propositional procedures are generated for each implementation and are XORed in order to verify their satisfiability. If this clause is satisfiable, then the implementations are not the same. In other words, the only way to output the value true in an XOR gate is when both inputs are not equal and, thus, not equivalent; otherwise, when both inputs are either true or false, the output is false. With structural-based techniques, different implementations are analyzed in order to identify structures that are similar between them, so that complex Boolean data structures or Boolean equations can be avoided [282]. Finally, signature-based techniques take into account logic simulation, so that the output generated at each node is the

signature of that node, and similarities between implementations are used to testify their equivalence.

Two examples of Equivalence Checking tools for hardware verification are *EQUIPE* and the tool from [283].

3.4.7 Static Analysis

Static Analysis was pioneered by Cousot [284] for abstract interpretation of software structures. From that work, numerous tools emerged, the first of all being Lint for static analysis of C programs. After the popularity of this tool, the term “linting” was coined. However, today this type of analysis does not apply only to software, but to HDL as well. Static analysis techniques range from the most mundane (statistics on the density of comments, for instance) to the more complex, semantics-based analysis techniques. Some tools provided by industry vendors are *Klee*, *Microsoft SAGE*, and *CompCert*. Klee is a static analyzer for C programs based on symbolic execution. According to [285], a C program is modeled as a binary tree for the analysis and Klee traverses the tree from root to leaves, generating sets of constraints at each decision state (e.g. conditionals and loops) until it finds an error or an exit state. When either is detected, Klee solves the generated constraint to create a test case that will be applied to the unmodified program. Also SAGE uses symbolic execution. It generates tests for software programs by choosing good candidates for the inputs of a program to optimize the constraints generated during the analysis phases [286]. Differently than Klee, SAGE is a machine-code based approach. This allows the tool to generate tests for programs regardless of their source language. It takes into consideration only the underlying system architecture. CompCert is a formally verified compiler for C programs, with special attention to safety-critical systems. CompCert’s goal is to generate optimized executable files that are free of miscompilation errors while observing the semantic preservation [264]. According to [287], the flow of CompCert is composed of twenty passes that cover the transformation from C source code to object code. These twenty passes are grouped in four phases. First, Parsing preprocesses the source files to generate an *Abstract Syntax Tree* (AST). The parser is automatically generated along with a proof of its correctness. Second, C front-end compiler checks the types inferred for the expressions and determines the order of execution. Third, Back-end compiler refines and optimizes the front-end’s output on the target architecture. Fourth, Assembling takes

the AST produced by the third phase and produces the final object and executable files using debugging information from the parser. The internal tool Valex helps to increase the confidence on the result by checking the generated executable files.

3.4.8 Semiformal Verification

Another methodology that is also widely employed by the industry is the combination of FM with simulation approaches. Simulation gives a good glimpse of the system functionality, as it needs inputs to exercise the architecture and produce outputs that can be verified against a golden model, to testify their functional correctness. However, the bigger the system, e.g., many input and output pins, many IP blocks, long chains, the longer it takes to evaluate the system's correctness. In addition, depending on the system size, checking all the possible input combinations and all the possible internal values using simulation is practically impossible, so the coverage is not complete. Tools that employ hybrid approaches between simulation and formal verification can mix them in various forms. An example is (Deep) Dynamic Formal Verification [288], where simulation is used to direct the architecture to a specific state, and from there, formal tools try to completely verify a subset of the state space.

3.4.9 Conclusion

Woodcock et al. [289] showed in a survey of industrial use of formal methods in 62 projects that the largest single application domain was transport (16%), followed by the financial sector (12%). The formal methods work helped to make the informal requirement specifications more precise. In addition, many errors were found during the proof activities. As a result, very few bugs were found during the testing of the systems. A remark was that using formal methods made the resulting software more correct, and that the costs tended to be increased early in the development life cycle, but reduced later. In general, the effect on development time, cost, and quality of the resulting product was generally positive. Several standards for the industrial development of safety-critical software even require the use of formal methods for the highest software safety integrity levels. We can conclude that there is no a single verification technique to completely verify an architecture. Since theorem proving does

not deal with states, but with formulas, it can be used with projects of any complexity. However, it demands a high degree of knowledge of the design under verification and of logic complex formulas, and it is difficult to automate. The main advantage of Model Checking is that it reports illegal paths (known as counterexamples), which supports the correction of bugs. However, since the number of states used to model the system grows exponentially with the number of variables, a common problem is state space explosion. SMC, BMC, and SMT are therefore good options only for the verification of parts of an architecture, and not for the whole. Equivalence Checking compares different levels of the same implementation to guarantee that they are functionally the same. This technique also checks the consistency of an optimization after its implementation or between different abstraction levels. Anyway, some caution is necessary in this case, as the correlation between these implementations is not direct. One of the main reasons for the scarce adoption of formal methods is the lack of the necessary mathematical background, as well as the lack of robust and user-friendly tool support. In Industry, simulation is still the most widely used method for verification and validation because it is easy to use and gives a very good idea of the system's behavior. However, when the number of input signals increases (as in large systems), it cannot cover exhaustively the state space. Often, a good solution is adopting hybrid approaches, which combine the superior coverage of formal verification, and the scalability of simulation. In conclusion, with carefully chosen FMs that meet requirements of application domains, and that are adapted accordingly, automated FMs are powerful enough to help to the development of high-quality ES.

Chapter 4

Proposed Method

In this chapter, we present the various steps that are adopted in studying our research problem.

4.1 Research Methodology

Methods for obtaining answers to professional questions range from the fairly informal, to the strictly scientific. Research is a key source of providing guidelines. When you undertake a research study to find out answers to a question, you are implying that the process being applied uses methods that have been deeply tested for their validity. The logical scheme that is used to obtain answers to scientific questions is called *scientific method* [290]. At this point, it seems appropriate to explain the difference between research methods and research methodology. *Research methods* are all the techniques that are used for conduction of research. Thus, they refer to the techniques the researchers use in performing research operations. *Research methodology* is a way to systematically solve the research problem. Thus, it may be understood as a science of studying how research is done scientifically [291]. The researcher needs to know not only the research methods/techniques, but also the methodology. In order to answer questions, we have to collect, analyze, and interpreting information,

but to qualify as research, the process must have certain characteristics: be controlled, rigorous, systematic, valid and verifiable, empirical and critical. Our research process starts with defining a research goal. As the aim of our research is to provide tools and methods that are relevant for the industrial practitioners, the above research goal is defined considering the state of the practice and the state-of-the-art literature, and based on industrial challenges that needs to be addressed. The initial point for defining our research goal is the engineering challenges that come from engineering process currently being used in real-world development of ES. In order to formulate a research goal, we perform a critical analysis of the relevant literature and practice. We make sure that the defined research goal has not been previously addressed in the existing body of knowledge. Later in the process, the literature review serves to consolidate our knowledge base and helps us to integrate our findings with the existing body of knowledge [292]. The formulation of a research problem is fundamental. A research problem may take a number of forms, from the very simple to the very complex. The way you formulate a problem determines the steps that follow in the research journey. In the next step, we propose a solution that addresses the identified research goal. During the last step of the research process, we perform validation of our research results. After the validation phase is concluded, the research results are summarized into a research manuscript.

4.2 Research Problem

This work aims to provide an approach to support the *design* and *automatic verification of redundant system architectures* with respect to the given specifications, while ensuring the fulfillment of fail-operational requirements. In this work, we tackle the following research question and present contributions to address it.

Given a high-level system model, how to perform the DSE to find an “optimum” redundant schema?

4.2.1 Existing Limitations

In most of the works we have founded in the literature, only a portion of the problem was actually explored, either by:

- addressing single-objective DSE problems
- applying simple and unrealistic constraints and objectives
- taking into account a fixed number of FT techniques
- referring to a homogeneous or custom architecture
- using a static scheduling algorithm for exploration
- assuming observable errors
- not considering mixed-critical fault management.

Furthermore, the typical crucial activities that the designer has to face, such as:

- extraction of models for the analysis of alternatives
- fast evaluation of properties (e.g., timing, power consumption, etc.)
- selection and comparison among alternatives

are not yet totally automated or currently supported by tool suites. To make things worse, there is no a single verification technique to completely verify an architecture, and there is scarce adoption of FM in industry.

4.2.2 Problem Formulation

The RAP we are facing is illustrated in Figure 4.1. Consider a non-redundant system $A = \{C_1, C_2, \dots, C_n\}$ consisting of n components, and suppose that for each component C_i a library of applicable redundant design patterns $lib_i = \{P_i^1, P_i^2, \dots, P_i^{m_i}\}$ is available. Given a function that maps each component C_i to a redundant pattern P_i^j such that $P_i^j \in lib_i$, defining a redundant extension of the system $A_R = \{P_1^{j_1}, P_2^{j_2}, \dots, P_n^{j_n}\}$, our aim is to find the function R that evaluates the reliability of each possible extension A_R of the architecture A , and compute the appropriate allocation of redundant design patterns to basic (non-redundant) system components that maximizes the system reliability and optimize some other non-functional parameters, given various system-level constraints.

The problem can be formulated as follows:

- given a (non-redundant) system $A = \{C_1, C_2, \dots, C_n\}$ defining the architecture and the behavior of a component based system, made of n components such that
- each component C_i has a (finite) library of applicable redundant patterns $lib_i = \{P_i^1, P_i^2, \dots, P_i^{m_i}\}$, and
- given a function that maps each component C_i to a redundant pattern P_i^j such that $P_i^j \in lib_i$, defining a redundant extension of the system $A_R = \{P_1^{j_1}, P_2^{j_2}, \dots, P_n^{j_n}\}$,
- our aim is to find a function R that evaluates the reliability of each possible extension A_R of the architecture A .

4.3 Overview of the Approach

In the following, we discuss the problem from an optimization perspective where system synthesis can be considered a constrained COP. We propose a *multi-objectives DSE* process that *automatically* selects the appropriate technique or set of FT techniques (at logical level) to be applied to the original non-redundant system to obtain a redundant one, optimizing simultaneously a collection of objective functions. With a *Model-driven* approach, complexities of ES are managed at the highest level of abstraction (i.e., system level) by using models as key artefacts throughout the development process. If these models are formally defined, the process can be automated. If the transformations from a model to another are formally defined, they preserve the equivalence (automatic verification). This task will be applied before the implementation phase and independently from the specific target platform, to be as more general as possible.

This methodology allows the designer to apply to a given heterogeneous system architecture various FT techniques (theoretically all existent ones), analyzing, evaluating and comparing the resulting redundant schemes in front of multiple optimization objectives, before mapping the system under study onto a physical architecture. Although non-functional requirements (like reliability or performance) are highly dependent on the implementation, the introduction of this abstract information at the early stages of a design can be of great benefit to the design process. First of all it gives an indication about the possible impacts of a considered redundant scheme. In addition, it is useful

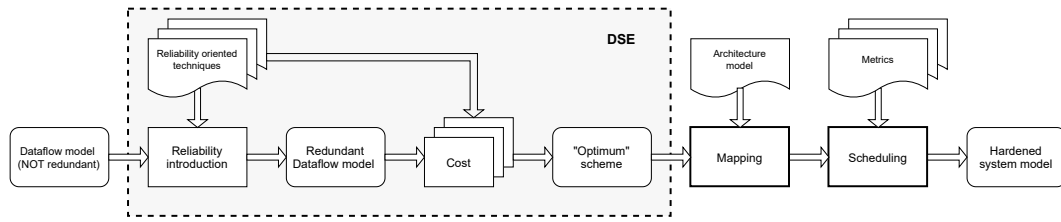


Figure 4.1: Redundant system-level synthesis flow

for discarding unsuitable design points and for reducing the design space of feasible candidates; especially in view of the following mapping activity which is compute intensive (see Figure 4.1). This helps reduce errors, lowering cost, shortening the design cycle, and automate the design process.

4.4 Input and Output

We consider as **input** of the process the following items.

- I1: **System model**: the data-flow model of the basic (non-redundant) system. It is represented by a directed graph where nodes represent the components of the architecture and edges describe how components are connected.
- I2: **Fault model**: specification of the failures to be handled. Components are equipped with sets of Boolean fault variables, which determine the behavior when one or more faults occur.
- I3: **Objectives**: maximizing or minimizing different kind of criteria.
- I4: **FT patterns**: library of redundant design patterns (Duplex, TMR, etc.).
- I5: **Design constraints**: the above model must allow the designer to specify constraints that the candidate scheme has to satisfy, like conform to size limits, budget power consumption, satisfy reliability requirements, and meet cost targets.

And as **output** the best redundant architecture that fulfills functional and fault management requirements on the basis of cost/performance data. See Figure 4.2.

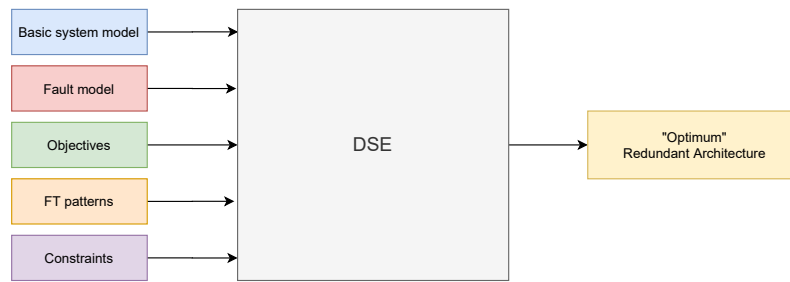


Figure 4.2: Inputs and outputs of DSE approach proposed

4.4.1 System Model

Usually, the system development process begins with the definition of the high-level system goals based on stakeholder inputs. From the analysis of those requirements, the specification will be defined. The specification says what things the system does, but it does not say how. Describing how the system implements those functions is the purpose of the architecture. The high level goals lead the development team to define a certain architecture style. This architecture drives a high-level model (I1). We consider this model as our starting point. It is a data-flow model. Data-flow modeling examines how data moves around the system, focusing on how things connect. Since the choice of representation affects both the storage and computational time to perform look-ups and algorithms, we represent the basic system as interconnected components, specifying for each node a list of neighbors, defining its logic through SMT variables and constraints. The main purpose of describing processes formally is that this allows us to subject the processes to formal analysis, i.e. it allows us to talk about their properties in a precise way. The building blocks of the system under analysis are represented in the theory of EUF. Namely, an uninterpreted function is used to represent the behavior of a component as a generic function over the reals.

4.4.2 Fault Model

We can then augment the original architecture model with fault information to characterize anomalous conditions, producing the architecture fault model (I2). A system *failure* occurs when the delivered service deviates from fulfilling the system's function. The occurrence of faults is modeled with the introduction of Boolean fault variables. Then, we build a model of the deviation (aka TLE) of the system under analysis from its nominal behavior. The model of the

deviation is an SMT formula with respect to the theory of EUF, written as an SMT(EUF) formula.

4.4.3 Objectives

The goal of ES design is, certainly, a system that will optimize various metrics of design. The objectives (I3) listed below are typically used as optimization goal of the overall system, can universally be applied to DSE and are not domain-specific.

- *Reliability*: to assess the reliability of the redundant architectures stored in the library, we express it relative to their composing elements. For example, the probability of failure of a TMR can be computed as follows.

$$P_{failure} = p_v + (1 - p_v)(3p_m^2 - 2p_m^3)$$

where p_v and p_m are the failure probability of the voter and the failure of probability of a module respectively. Once the failure probability is found, reliability is trivially obtained by complementing it. If the three modules are different, or more in general, their failure probabilities are different, the formula is more complex and can be obtained by directly reasoning on the MCS, which in the case of a TMR are the following.

$$MCS = \{\{m_1, m_2\}, \{m_1, m_3\}, \{m_2, m_3\}, \{v\}\} \quad (4.1)$$

where m_1 , m_2 , and m_3 are the three modules and v is the voter. The resulting formula is the sum of the products of the probabilities associated to each CS.

$$\begin{aligned} P_{failure} = & p_{m_1}p_{m_2}(1 - p_{m_3})(1 - p_v) + \\ & p_{m_1}(1 - p_{m_2})p_{m_3}(1 - p_v) + \\ & (1 - p_{m_1})p_{m_2}p_{m_3}(1 - p_v) + \\ & (1 - p_{m_1}) * (1 - p_{m_2}) * (1 - p_{m_3}) * p_v \end{aligned}$$

Employing those redundant patterns, we can compute the reliability of the entire system as illustrated in the next sections, and thereafter evaluate the reliability improvement for the redundant architectures as the improvement in system reliability compared to the basic (not redundant) system.

- *Cost*: the cost of a design could be measured as the sum of all components. At the level of abstraction we are reasoning, we have no information about the specific component that will be implemented, so we cannot base on the wholesale prices from different component manufacturers. So far, for each redundant pattern applied, the cost can be expressed as a percentage increase with reference to the basic component. For instance, the cost of a TMR pattern has a recurring cost of 300% comparing to the basic system, due to the using of three parallel modules. We can ignore the cost of voter which is normally a simple hardware circuit that depends on the type of the output control signal. And there are no additional development costs because the three modules are identical and they use the same software.
- *Power dissipation*: as for cost, power dissipation of a redundant pattern can be expressed as a percentage increase compared to the basic (not redundant) component. For instance, the power dissipation of a TMR is 300% the power dissipation of the basic component (ignoring the dissipation of the voter).
- *Size*: introducing redundant modules means also increasing the area occupied by the system. For instance, a TMR occupies three times the area of the basic module (increase of 300%).
- *Weight*: as for size, weight of a redundant pattern can be expressed as a percentage increase compared to the basic (not redundant) component. For instance, the weight of a TMR is 300% the weight of the basic component (ignoring the weight of the voter).
- *Execution time*: in general, the speed of a design can be expressed by different metrics (such as throughput, amount of data processed, response time for certain events, etc.). Also in this case, we express it by comparing the redundant scheme and the basic module. For instance, the TMR pattern has a little influence on the execution time comparing to the basic system, since the three modules are running separately in parallel and we can ignore the small delay introduced by the voter. More in general, the execution time depends on the configuration of the subsequent components (series/parallel) and it is influenced by the slowest one.

4.4.4 Redundant Patterns

To construct the library of FT design patterns (I4), widely used and proven solutions in the field of ES have to be collected from literature. These solutions should be generalized establishing a high-level and abstract representation that can be used to evaluate the impact of the collected solutions when applied to the basic system architecture, independently from a specific application or a specific implementation.

4.4.5 Design Constraints

As additional input to the proposed framework, we consider design constraints (I5) - specified by the designers - that the generated architectures have to satisfy. The design constraints can have a local or global scope, concerning either individual components or the entire system.

The redundancy allocation system can be formally defined by a 4-tuple:

$$S = \langle R, C, P, A \rangle$$

where:

- R is the set of available redundant patterns;
- C is the capacity function, characterizing the number of available identical units from each pattern type;
- P denotes the set of the system components;
- A is the redundancy allocation function associating every component with the redundant pattern required for objectives optimization.

At the end, we will obtain a set of possible solutions. Obviously, the designer is might interested in one single solution to deploy. To this end, the designer can choose just a single point among the set, for example on the basis of the priority assigned to the objectives.

4.5 Contributions

We propose a DSE framework that supports the activity of discovering and evaluating design alternatives, captured by design specifications, during system development. We extend the work proposed by Bozzano et al. [154] along

several directions, in order to propose a fully automated approach to the reliability assessment of complex redundant architectures. First of all, Bozzano et al. study the reliability of a *given* redundant architecture. Instead, we deal with a DSE process in which design alternatives are evaluated to *optimize* the architecture. Our method includes the reliability assessment as part of the evaluation of alternative designs. Moreover, we introduce a novel assessment of reliability based on configuration and fault variables, presented below, that affects the structure of the BDD, pushing it to work in a kind of deductive manner. Furthermore, Bozzano et al. adopted TMR as redundancy architecture, while we build a library of redundancy architectures applicable to the basic components, dealing with the complexities related to their selection and connections. They implemented the approach by leveraging several existing tools: they used the OCRA language [293] to specify the architecture under analysis, and employed the language SMV [155] and its associated tool XSAP [156] for the symbolic representation of the system. In our case, architecture modeling is performed completely using SMT. We leverage the power of SMT techniques to automate the reasoning part, by representing the architecture as uninterpreted functions, and extracting all the deviation conditions resolving an AllSMT problem.

The key point of our work is the *symbolic encoding* of the reliability search problem. Given an abstract description of the architecture, the approach automatically extracts a symbolic reliability function mapping the probability of fault of the basic components to the probability that the overall architecture deviates from the expected behavior. We encode the problem with a symbolic technique that allows us to compare different redundant architectural configurations independently of the specific values of failure probability, as well as evaluate different components that implement the same architecture.

Furthermore, together with reliability, we also consider other non-functional requirements such as cost, power dissipation, size area, and execution time, to find the assignments of redundant design patterns to basic components that optimize (minimize/maximize) some objective functions, addressing therefore a MOOP. At system level, estimation of non-functional properties is particularly difficult as the sub-components are not designed yet, and the individual tasks of the application may not be fully specified.

Lastly, we also integrate the exact SMT-based method with a heuristic that can help solve the optimization problem when the sheer size of the design space makes enumerating every design point prohibitive.

Besides, we introduce some refinements and smart solutions in every step of our method, in order to further improve the performance.

Our contribution is the following:

- Providing a high-level description of system architecture in terms of connections between individual components.
- Formalizing the constraints that the generated architectures have to satisfy, and not domain-specific objectives that can universally be applied to DSE.
- Building a library of redundant design patterns.
- Defining a novel symbolic encoding of reliability function, by encoding the derived optimization problem into a SMT form, and generating a formula that represents all valid redundant architectures for the given system, employing the patterns contained in the library.
- Evaluating additional non-functional requirements, supporting the DSE wrt. multiple objectives.
- Defining an approach to support the design of redundant system architectures, providing an exact and an approximate methods.
- Implementing the proposed method with PySMT [294], an open-source python library that provides a solver-agnostic interface.

With a *Model-driven* approach, complexities of ES are managed at the highest level of abstraction (system level) by using models as key artefacts throughout the development process. We reason at logical level, applying the proposed method before the implementation phase and deployment phase, thus independently from the specific target platform, to be as more general as possible. Our underlying aim is to provide results that show that an application of the proposed SMT-based method to solve DSE is promising and valuable against well-known evolutionary-based approaches.

4.6 Challenges

While existing model-driven frameworks are able to explore the design space of smaller problems by exhaustively traversing reachable states and checking

global constraints and goals in each state, our approach also wants to define rules for guidance that help the exploration and addresses the following challenges:

- **Extensibility:** we want to define a framework as much general as possible and easy to use that can be applicable on different design problems; the set of criteria should be extensible for adapting the approach to various domains.
- **Decisions identification:** during exploration, the framework should distinguish the guidance from the exploration strategy to easily allow the modification of both parts. We can combine the advantages of high level rules that guide the search, using consolidated techniques, with those of the solver, i.e., only obtaining feasible implementations.
- **Traversed design space reduction:** the guidance should reduce the number of traversed states before finding solutions and ensure that no valid solutions are removed by the cut-off criteria.
- **Provide optimal solutions:** the guided framework should find the solutions that are (near-)optimal with respect to a user-defined metric. Moreover, it should be able to find other (less optimal) solutions if necessary.
- **Cover dynamic aspects:** usually, existing methods target static aspects that only cover scenarios which consider only one system state at a time. We want the proposed method to allow for an analysis of dynamic aspects of models, i.e. the evolution of system states. This means that faults are associated with dynamics. Please note that this aspect is not covered in this work. We reserve it for future work.

The framework automatically selects the appropriate technique or set of fault-tolerant techniques (at logic level) to be applied to the original system to obtain a redundant one, optimizing simultaneously a collection of objective functions. It will explore and analyze several functionally equivalent alternative designs. It is based on a hybrid optimization approach that combines exact techniques with a heuristic algorithm. Specifically, a SMT solver determines feasible solutions of relaxed problems, and those solutions are used as a starting point for a metaheuristic search. The key feature of this approach is

that it combines the advantages of the metaheuristic, i.e., being able to consider multiple and non-linear design objectives, with those of the solver, i.e., only obtaining feasible implementations. We will not create a new technique, but rather create high level rules that guide the search, using consolidated techniques

Chapter 5

Reliability Assessment of Redundant Architectures

In the following we lay out the theoretical foundations of the encoding problem. We recall the method presented by Bozzano et al. [154], adding some improvements, and presenting main extensions.

5.1 Assumptions

Most hardware faults are random and result from physical defects, either sustained during manufacturing or developed over time as components wear out or suffer shocks from the surrounding physical world. Software faults, on the other hand, are not physical; software does not wear out. Software faults result from the invocation of software paths that contain defects in the software design or implementation. The following assumptions are made.

- We consider our software as black boxes (higher level of abstraction) and we assume that there are no design errors.
- We assume that FT patterns are not limited, i.e., we can use more patterns of the same kind, if not specified from a design constraint.

- We do not consider component mixing, i.e. a mix of components within a subsystem is not allowed, in other words, for each component a single pattern is allocable for redundancy.
- Basic events are independent, and each component is critical, i.e., its failure triggers the TLE.
- The single failure probabilities of the modules composing the components are given and expressed relative to a certain time interval. We also assume that such failure rate is constant in the life-cycle of each component. This is a fair assumption, as usually reliability standards report this information in terms of MTTF or MTBF.
- A failure leads a component to behave incorrectly and produce an arbitrary output.
- We refer to a coherent (or monotone) model, i.e., for each scenario where the system fails, adding more failure events maintains the failure condition.
- For simplicity, we assume that all redundancy is active. In this case, redundant components automatically pick up load on failure. They do not have to detect component failure and do not have to switch to redundant resource. Furthermore, a failed component cannot return to a working state (it is not repairable).
- To leverage analytical methods, we also assume that the redundant components preserve the original component's interface. Moreover, we do not consider hierarchical models, but only flat models.

5.2 Modeling the System Architecture

Consider a high-level system architecture represented by a DAG, in which nodes represent the components of the architecture (denoted by C_i) and edges describe how components are connected. Figure 5.1 shows three different architecture graphs for system composed of six components, respectively connected in series, parallel, and complex configurations. A DAG is formed by vertices and by edges going from one node to another. These edges are directed, which means that they have an orientation. It is also acyclic, which means that there

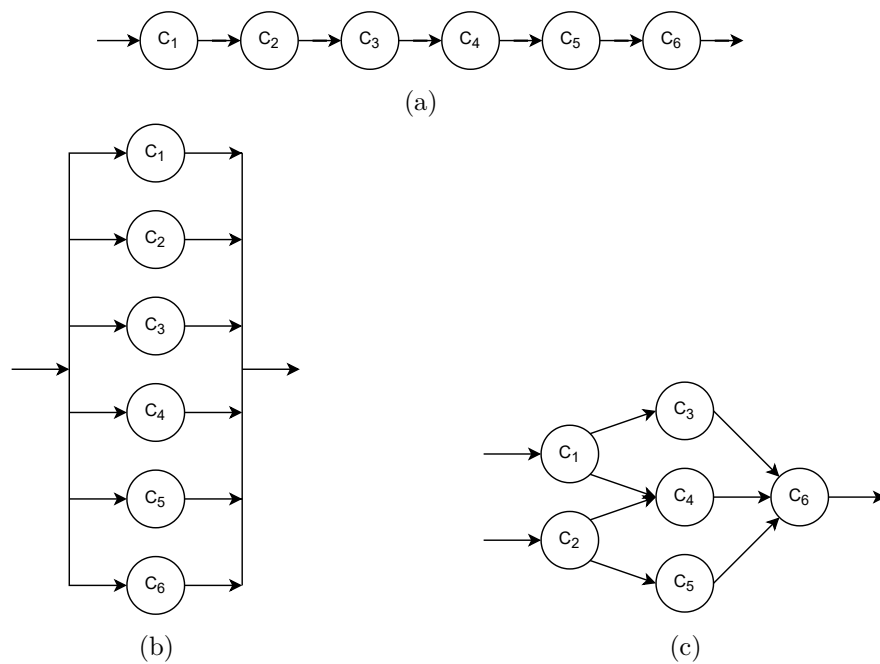


Figure 5.1: Example of series (a), parallel (b), and complex (c) system architectures.

are no feedback loops, i.e., a variable cannot be its own descendant. A *topological ordering* of a DAG is an ordering of its vertices into a sequence, such that for every edge the start vertex of the edge occurs earlier in the sequence than the ending vertex of the edge. A DAG is a suitable representation of a component-based system, since data enters a processing element through its incoming edges and leaves the element through its outgoing edges. Data-flow programming languages describe systems of operations on data streams, and the connections between the outputs of some operations and the inputs of others.

Each component is equipped with a Boolean fault variable that determines the behavior of the component when one or more faults occur.

Definition 1 (Fault variable). *A fault variable F_i is a Boolean variable that determines the behavior of a component when a fault occurs.*

Each computing module within the component has two separate behaviors: nominal (M_N) and faulty (M_F), both represented as a generic function over the reals, using uninterpreted functions. With the theory of EUF, function symbols have no specific property, except for the fact that they are functions. Since the functions are uninterpreted, they allow us to express the functional

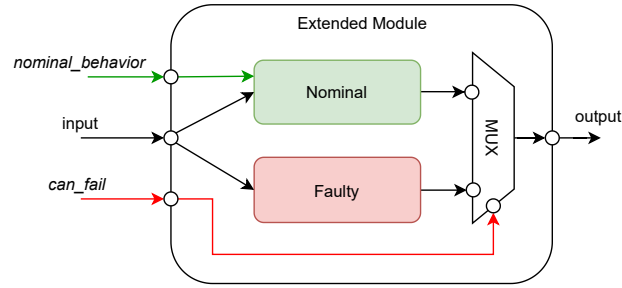


Figure 5.2: Example module with nominal and faulty behavior

properties of the behavior of components abstracting from the specific implementations. The outputs of each pair of nominal and faulty modules are provided to a multiplexer, which selects the proper signal according to the fault event. Denoting by F_M the probability of fault of a given module, the formal model that describes the setting shown in Figure 5.2 is defined using the following SMT formula.

$$\neg F_C \implies output = nominal_behavior(input). \quad (5.1)$$

Others elements such as Voters, Comparators, and Fault-detectors have well defined implementations, and they do not need therefore to be modeled with an uninterpreted function. Each module receives as parameters: the input representing the input values of the computation (of type real), a Boolean parameter *can_fail* that enables the component to have internal failures, and the *nominal behaviour* of the computation, which is a function modeling the computation in the nominal case. In addition, it has a local variable *is_faulty* that keeps track of the current behavior (nominal or faulty). Voters, Comparators, and Fault detectors receives as parameters: the input values (of type real), the Boolean parameter *can_fail*, which enables the component to have internal failures, and the local variable *is_faulty* that keeps track of the current behavior. In short, we extend each module with a fault model, getting an *Extended Module* (EM). For example, Figure 5.3 shows the modeling technique described above applied to a TMR. For each sub-component, a multiplexer chooses nominal or faulty output depending on the *can_fail* parameter.

Given an input i and naming i_{m_i} and o_{m_i} respectively the input and output of the module m_i , and modeling the nominal behavior of the module m_i with the uninterpreted function $beh_{m_iN}(i)$, and its faulty behavior with the uninterpreted function $beh_{m_iF}(i)$, the outputs of the module m_i depend on both

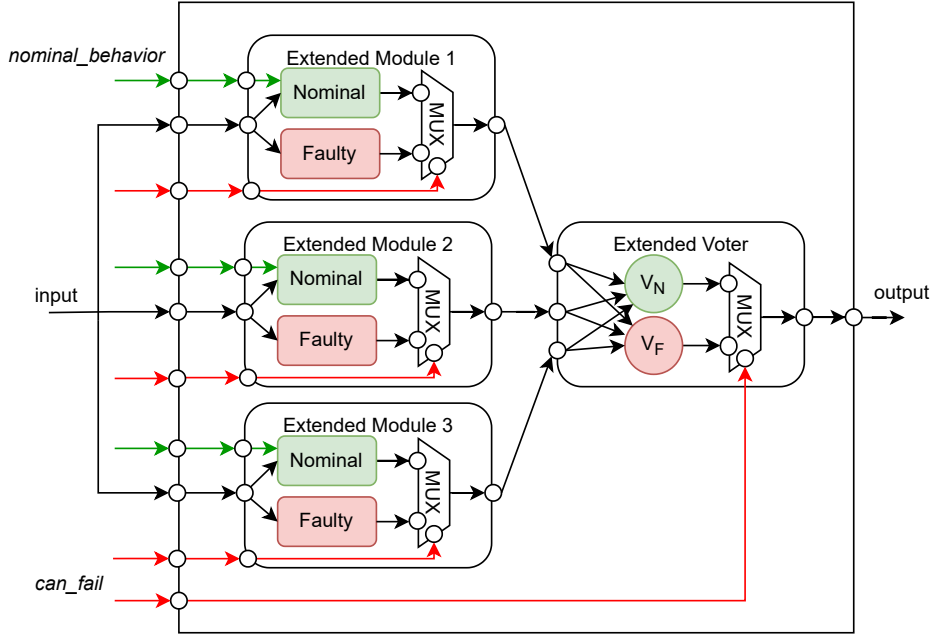


Figure 5.3: Example of redundant pattern modeled with nominal and faulty behavior

its inputs and its internal behavior. Thus, it can be modeled using a SMT formula as follows:

$$Formula_M := ITE \left(can_fail, \left(\begin{array}{l} ITE(F_{M1}, o_{M1} = beh_{M1F}(i_1), o_{M1} = beh_{M1N}(i_{M1}) \wedge \\ ITE(F_{M2}, o_{M2} = beh_{M2F}(i_2), o_{M2} = beh_{M2N}(i_{M2}) \wedge \\ ITE(F_{M3}, o_{M3} = beh_{M3F}(i_3), o_{M3} = beh_{M3N}(i_{M3}) \end{array} \right), \left(\begin{array}{l} o_{M1} = beh_{M1N}(i_{M1}) \wedge \\ o_{M2} = beh_{M2N}(i_{M2}) \wedge \\ o_{M3} = beh_{M3N}(i_{M3}) \end{array} \right) \right) \quad (5.2)$$

We rely on real data types in order to describe infinite domain values, and the uninterpreted functions we refer to have therefore both real domain and co-domain. The real output o_{mi} is constrained only if it cannot fail (i.e. can_fail is False or the fault variable of the module is False), otherwise the output takes an arbitrary value. Please note that, since the faulty behaviour of a module consists in generating a random real value, the arbitrary output could also assume the nominal value.

While the internal behavior of a computing module is abstracted by using uninterpreted functions, the voter has a well defined nominal implementation, as stated above, and it can be modeled by the following SMT expression:

$$Formula_v := (\neg can_fail \vee (can_fail \wedge \neg F_v)) \rightarrow \left(\begin{array}{l} o_{M1} = o_{M2} \vee o_{M1} = o_{M3} \rightarrow o_v = o_{M1} \wedge \\ o_{M2} = o_{M3} \rightarrow o_v = o_{M2} \wedge \\ o_{M1} \neq o_{M2} \wedge o_{M1} \neq o_{M3} \rightarrow (o_v = o_{M1} \vee o_v = o_{M2} \vee o_v = o_{M3}) \end{array} \right) \wedge \quad (5.3) \\ (can_fail \wedge F_v) \rightarrow beh_{VF}(o_{M1}, o_{M2}, o_{M3})$$

Formula 5.3 states that if the pattern cannot fail or it can fail and the voter is not faulty then the voter behaves correctly, and the output value o_v corresponds to the majority value of its inputs (which is the output of the three modules). If all the three inputs are different, then the voter chooses a random input value and outputs it. If the voter is faulty instead its behavior is unpredictable, and it is modeled by the 3-ary unconstrained uninterpreted function $beh_{vF}(i_0, i_1, i_2)$ that produces an arbitrary real value (this is managed by the tool, which we use as black box). The formula that models the entire TMR can be expressed by combining equations 5.2 and 5.3.

$$Formula_TMR := Formula_v \wedge Formula_m \quad (5.4)$$

```

1 class TmrV111(Pattern):
2
3     n_f_atoms = 4
4
5     def __init__(self, comp_name: str, comp_n_inputs: int, modules_fault_atoms: list,
6                 voter_fault_atom: Symbol, nominal_mod_beh: Symbol):
7
8         """
9         :param comp_name: name of basic component
10        :param comp_n_inputs: number of inputs of the basic component
11        :param modules_fault_atoms: fault atoms for the 3 modules
12        :param voter_fault_atom: fault atom for the voter
13        :param nominal_mod_beh: nominal behaviour
14        """
15
16        pattern_name = comp_name + "." + PatternType.TMR_V111.name
17        modules = [FaultyModule(pattern_name + ".M" + str(idx), comp_n_inputs,
18                               modules_fault_atoms[idx], nominal_mod_beh) for idx in range(3)]
19        modules_out_ports = []
20
21        # The output of the modules are the inputs of the voter
22        for module in modules:
23            modules_out_ports.extend(module.output_ports)
24            assert len(modules_out_ports) == 3, "[" + pattern_name + "]" The voter must have 3
25            inputs"
26
27        self._voter = Voter(pattern_name + ".V", voter_fault_atom, input_ports=
28                            modules_out_ports)
29
30        # Output port of the pattern corresponds to the output port of the voter
31        output_ports = self._voter.output_ports
32        super(TmrV111, self).__init__(pattern_name, PatternType.TMR_V111, modules_fault_atoms
33                                    + [voter_fault_atom], modules, output_ports)
34
35        # Define behaviour formula: And of subcomponents behaviours
36        # Modules
37        subcomp_beh_formula = [module.behaviour_formula for module in modules]
38        # Voter
39        subcomp_beh_formula.append(self._voter.behaviour_formula)
40        self._behaviour_formula = And(subcomp_beh_formula)

```

Listing 5.1: TMR pattern definition

Listing 5.1 shows the Python code implementing it. Listing 5.2 presents the definition of the voter with three inputs.

```

1 class Voter(Component):
2
3     def __init__(self, name: str, fault_atom: Symbol, input_ports: list = None,
4                 output_port: Symbol = None):
5         """
6         :param name: name of voter
7         :param faulty_atom: symbol used to indicate whether the voter is faulty
8         :param input_ports: list of symbols corresponding to the voter's input ports
9         :param output_ports: list of symbols corresponding to the voter's output ports
10        """
11
12        self._fault_atom = fault_atom
13
14        # Define input and output ports
15        if input_ports is None:
16            input_ports = [Symbol(name + ".i" + str(idx), REAL) for idx in range(3)]
17        else:
18            assert len(input_ports) == 3, "Voter can only accept 3 input ports"
19        if output_port is None:
20            output_port = [Symbol(name + ".o0", REAL)]
21
22        super(Voter, self).__init__(name, ComponentType.VOTER, input_ports, output_port,
23                                   fault_atoms=[fault_atom])
24
25        # Define nominal behaviour
26        nom_behaviour = And(
27            Ite(
28                Or(
29                    Equals(self._input_ports[0], self._input_ports[1]),
30                    Equals(self._input_ports[0], self._input_ports[2])
31                ),
32                Equals(
33                    self._output_ports[0], self._input_ports[0]
34                ),
35                Equals(
36                    self._output_ports[0], self._input_ports[1]
37                )
38            )
39
40        # if faulty atom is false, then the behaviour is nominal
41        self._behaviour_formula = Implies(Not(self._fault_atom), nom_behaviour)

```

Listing 5.2: Voter used in the extended TMR

In order to help figure out this abstract representation, Listing 5.3 reports a simple test program for the TMR, and Figure 5.4 illustrates the output.

```

1 # Test - Example
2 if __name__ == "__main__":
3
4     nominal_beh = Symbol("nom-beh", FunctionType(REAL, [REAL]))
5     tmr = TmrV111("TMR_V111_A", 1, [Symbol("F0"), Symbol("F1"), Symbol("F2")], Symbol("F3"),
6                    nominal_beh)
7     print(tmr.behaviour_formula.serialize())

```

Listing 5.3: Test example for TMR

```

(((! F0) -> (TMR_V111_A.TMR_V111.M0.o0 = nom-beh(TMR_V111_A.TMR_V111.M0.i0))) &
(! F1) -> (TMR_V111_A.TMR_V111.M1.o0 = nom-beh(TMR_V111_A.TMR_V111.M1.i0))) &
(! F2) -> (TMR_V111_A.TMR_V111.M2.o0 = nom-beh(TMR_V111_A.TMR_V111.M2.i0))) &
(! F3) -> (((TMR_V111_A.TMR_V111.M0.o0 = TMR_V111_A.TMR_V111.M1.o0) |
(TMR_V111_A.TMR_V111.M0.o0 = TMR_V111_A.TMR_V111.M2.o0)) ? (TMR_V111_A.TMR_V111.V.o0 = TMR_V111_A.TMR_V111.M0.o0) :
(TMR_V111_A.TMR_V111.V.o0 = TMR_V111_A.TMR_V111.M1.o0)))

```

Figure 5.4: Output of the test example for TMR

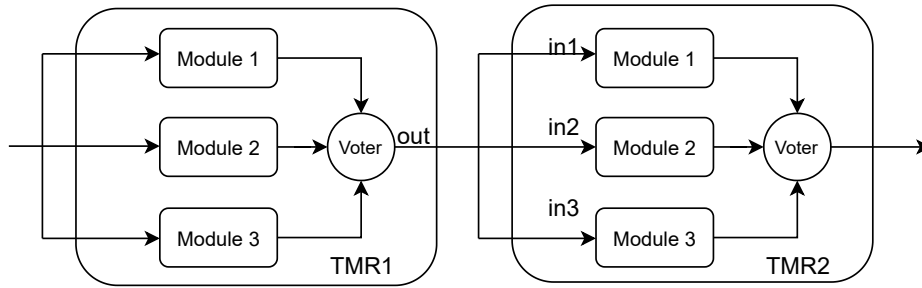


Figure 5.5: Linking constraints determine the connections between two components

This representation can be extended to any redundant patterns. Leveraging this method, the redundant architecture can be expressed through an SMT formula where the edges between two components (i.e., connections between two redundant patterns) can be expressed by equaling the output of the first component to the input of the subsequent one.

Definition 2 (Linking constraint). *A linking constraint Lnk_{ij} is an SMT formula that determines the connections between a redundant pattern p_i and the descendant redundant pattern p_j , by specifying the equalities among the outputs of the former and the inputs of the latter.*

For example, if we have two TMRs connected in series as in Figure 5.5, namely TMR1 and TMR2, we can specify how they are connected imposing the following constraints:

$$(TMR1.out = TMR2.in1) \wedge (TMR1.out = TMR2.in2) \wedge (TMR1.out = TMR2.in3)$$

5.3 Modeling the Miter

Modeling the modules with nominal and faulty behaviours gives us the possibility to describe both reference and faulty systems. The reference architecture is instantiated by providing `False` as `can_fail` parameter to all components, while the faulty description is obtained by setting it to `True`. This system composition is a selective switch (aka Miter [295]). By providing the same inputs to two architectures, we can compare them by evaluating the difference in the outputs. A deviation of the system under analysis from its nominal behavior is a TLE. The miter composition for an architecture composed of six

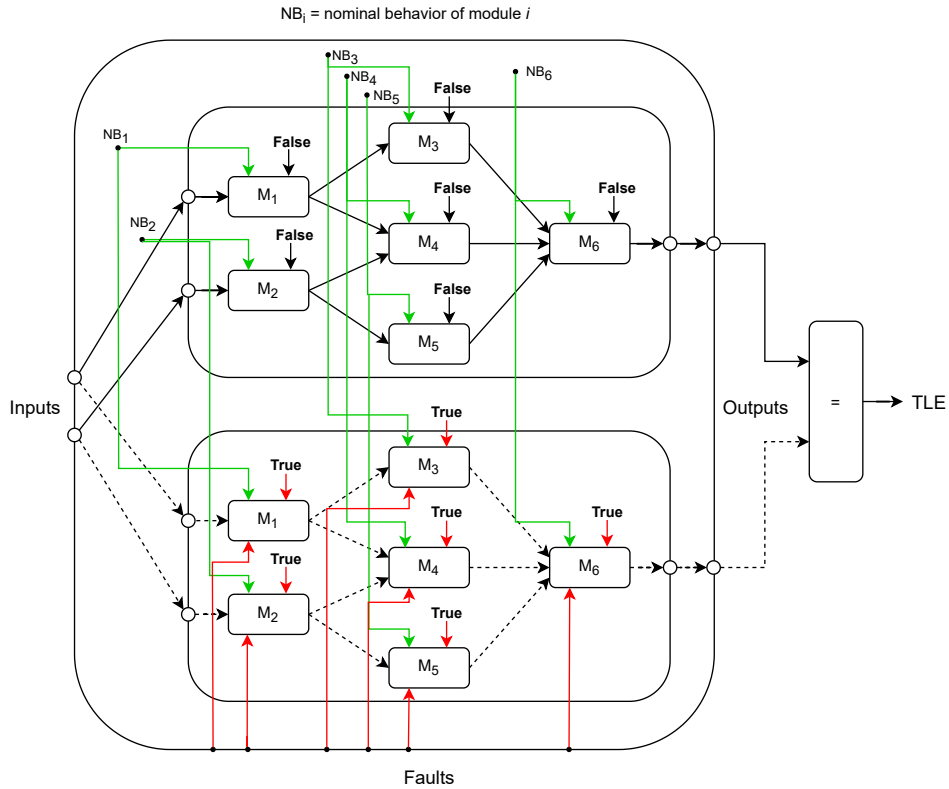


Figure 5.6: Miter composition

components like the one in Figure 5.1c is illustrated in Figure 5.6. If the model has a single output, then the TLE is the deviation between the two copies, if there are multiple outputs, then the TLE is the disjunction of all the output deviations:

$$TLE \stackrel{\text{def}}{=} \bigvee_{o \in \vec{O}} (o \neq o'). \quad (5.5)$$

The miter allows us to enable or disable the possibility to have faulty behaviors on the entire architecture. The miter composition can be generalized using the following formula:

$$patterns_behavior \wedge linking_constraints \wedge TLE \quad (5.6)$$

where *patterns-behavior* is the conjunction of the behaviour's formulas of both nominal patterns and faulty patterns, *linking-constraints* are the equalities that indicate how patterns are connected and *TLE* is the formula defined in 5.6. Since every (fallible) component has a corresponding (infallible) component in the reference architecture, we can compose a miter in which the

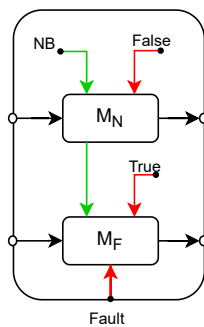


Figure 5.7: A stage aggregates nominal and faulty behaviors

corresponding components (faulty and reference) are tightly aggregated. This aggregation is called a stage (see Figure 5.7), and the resulting *stage-based miter* is illustrated in Figure 5.8. Although the two miters are logically equivalent, the latter combines reference and faulty components in the same block, allowing to emphasize the localization of the deviation of a component from its nominal behavior. Instead of quantifying out non-Boolean variables through a global AllSMT, modules are abstracted and quantifier elimination is performed individually on them. The outcomes of this procedure are then quantified and combined together by means of an efficient BDD-based procedure.

Every assignment that evaluates the formula 5.6 to True corresponds to a condition that may cause the two systems to provide different outputs.

5.4 Minimal Cut-sets Computation

In particular, we are interested in finding only the set of assignments to the fault variables F_i that are sufficient to trigger the TLE. Since the number of (Boolean) fault variables is finite, and their domain is finite and discrete, the number of such assignments is finite. In other words, we want to find a Boolean formula that consists only of fault variables and whose models correspond to the CSs of the architecture under analysis. Every CS can be represented, via a propositional formula, as a conjunction of component faults, and the set of configurations as a disjunction of CSs. If we represent the Miter as an SMT formula over input ports \vec{I} , output ports \vec{O} , fault variables \vec{F} , and TLE , the formula describing the CSs can be obtained via *quantifier elimination* as follows [154]:

$$\exists \vec{I}. \vec{O}. (\pi(\vec{I}, \vec{O}, \vec{F}) \wedge TLE(\vec{O})). \quad (5.7)$$

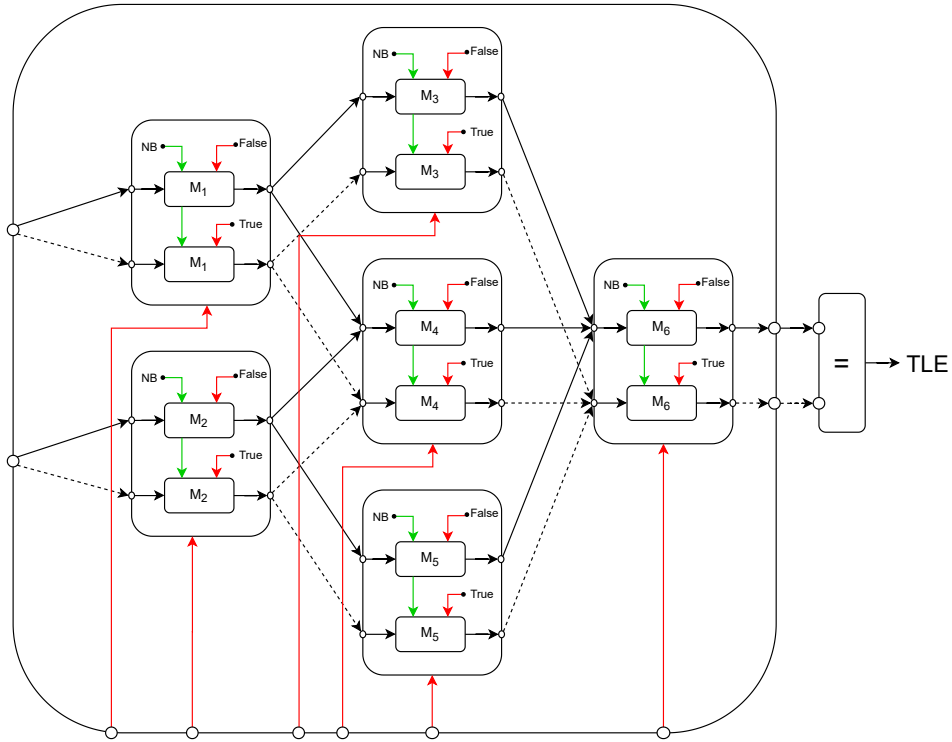


Figure 5.8: Stage-based Miter composition

Quantifier elimination consists in transforming a quantified formula into an equivalent quantifier-free formula. For example, in case of a TMR, on the basis of the equation 4.1, the set of CSs can be modeled with the following DNF formula:

$$F_v \vee (F_{m0} \wedge F_{m1}) \vee (F_{m0} \wedge F_{m2}) \vee (F_{m1} \wedge F_{m2}) \quad (5.8)$$

Please note that this formula implies all the other failure conditions, i.e., a solution of the MCS formula is also a solution of the formula representing other CSs. Formula 5.6 consists of Boolean variables, uninterpreted functions, and Real variables. It represents the set of assignments to the fault variables such that there exists an assignment to the inputs that allows the two architectures to provide different output values. Since the miter formula consists of input variables, output variables, and fault variables, quantifying out the inputs and outputs of each sub-component, we obtain a Boolean formula with only fault variables. An equisatisfiable formula consisting only of Boolean variables can be obtained through AllSMT, specifying that all theory variables have to be eliminated and only Boolean variables have to remain in the formula. The problem of extracting the CSs can be therefore encoded as an AllSMT for the

theory of EUF, i.e., computing all minimal solutions with respect to the set of decision variables.

5.5 Reliability Assessment

The formula representing the CSs can be converted into a BDD-based representation. Every assignment of Boolean variables determines a path from the root to a leaf. A path from the root to a True-leaf leads to TLE. A path from the root to a False-leaf, instead, represents an assignment that does not satisfy the configuration constraints or does not cause the TLE. Under the assumption that the events encoded by each fault variable are independent, the reliability function can be extracted by associating to each fault-variable a probability value, and computing the likelihood of the overall truth of the formula represented by the OBDD. From the OBDD we can calculate the fault probability of the entire system by recursively applying the following formula to each node n of the BDD:

$$\begin{cases} 1 & \text{if } n = \top \\ 0 & \text{if } n = \perp \\ F_i \cdot BddProb(n_{\top}) + \\ (1 - F_i) \cdot BddProb(n_{\perp}) & \text{if } n = ITE(F_i, n_{\top}, n_{\perp}) \end{cases} \quad (5.9)$$

where $BddProb(n)$ is the probability of failure considering the sub-tree of the OBDD rooted in node n , f_i is the failure probability of the component i associated to the node n , and n_{\top} and n_{\perp} denote the high node and the low node with respect to n . Basically, equation 5.9 means traversing the OBDD using a *Depth-First Search* (DFS) considering all paths that lead to True-leaves (a False-leaf corresponds to a probability of 0). For example, the BDD equivalent to the equation 5.8, representing the CSs of a TMR, is illustrated in Figure 5.9. Employing the recursive algorithm defined in equation 5.9, we obtain the following *failure probability function*:

$$f_{TMR} = f_v + (1 - f_v)(f_{m0}(f_{m1} + (1 - f_{m1})f_{m2})(1 - f_{m0})(f_{m1}f_{m2})) \quad (5.10)$$

Once the failure probability f_{ARCH} of a given system has been extracted, the *reliability function* can be trivially obtained by complementing it:

$$R_{arc} = 1 - f_{ARCH} \quad (5.11)$$

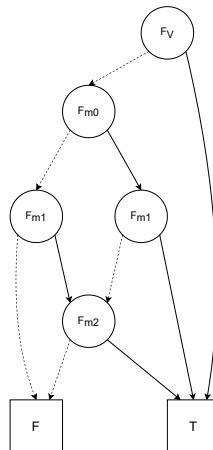


Figure 5.9: OBDD of the formula

5.6 Improvements and Refinements

Since the number of satisfiable assignments of the input formula grows with the number of variables, quantifier elimination can be slow. Worst-case complexities for useful theories tend to be towers of exponentials. This phase is therefore the main computational bottleneck of the method. In the following we propose a couple of solutions in order to overcome this issue and improve the performance of the algorithm.

5.6.1 Minimal Cut Sets Computation via Predicate Abstraction

First of all, we use the method introduced by Bozzano et al. [154] to abstract the behavior of redundant pattern via *predicate abstraction* in order to obtain a pure Boolean formula for modeling each possible deviation from its reference behavior. Since performance of AllSMT problems are related to the number of CSs, and this enumeration can be highly expensive, we rely on a suitable predicate abstraction, so that a unique, SMT-based quantifier elimination can be transformed into a BDD-based quantification on a boolean formula. The optimized method for CS construction leverages the correspondence in the miter between nominal and faulty modules. By applying a predicate abstraction on input and output ports of each stage, we characterize the ways in which the outputs of the component can deviate from the nominal case, given the internal faults and the deviations in the inputs deriving from upstream faults. The abstraction is explicitly represented by means of additional components

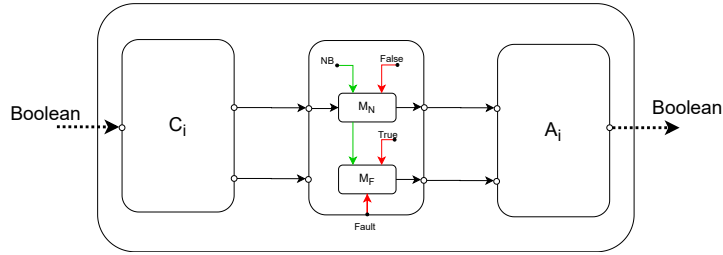


Figure 5.10: Abstract stage (aka CSA)

(named *concretizers* and *abstactors*) connected to the input and output ports of each stage of the architecture that allow us to abstract their behavior with a Boolean formula. The so called *abstract stage* is therefore the sequential composition of a concretizer, a stage, and an abstactor, linked by means of linker constraints. We call it CSA and it is illustrated in Figure 5.10.

The concretizers C_i receive as input an assignment to the predicates, and provide as output an instance of concrete signals satisfying them. Analogously, the abstactors A_i give as output the assignment to the predicates corresponding to the concrete data in input. More formally, an abstactor is a component with nominal and faulty Real inputs $\vec{I} \cup \vec{I}'$, in which every faulty input $in'_i \in \vec{I}'$ has a corresponding reference input $in_i \in \vec{I}$, Boolean outputs $\vec{O} = \{out_0, out_1, \dots, out_n\}$, and an internal behavior π_A that can be modeled with the SMT formula in equation 5.12.

$$\pi_A = \bigwedge_{i=0}^{len(\vec{I})} out_i \leftrightarrow (in_i = in'_i) \quad (5.12)$$

In like manner, a concretizer has Boolean inputs $\vec{I} = \{in_0, in_1, \dots, in_n\}$, nominal and faulty Real outputs $\vec{O} \cup \vec{O}'$, in which every faulty output $out'_i \in \vec{O}'$ has a corresponding reference output $out_i \in \vec{O}$, and a behavior π_C that can be modeled with the SMT formula in equation 5.13.

$$\pi_C = \bigwedge_{i=0}^{len(\vec{O})} in_i \leftrightarrow (out_i = out'_i) \quad (5.13)$$

The resulting architecture, depicted in Figure 5.11, has the same interface as the concrete one by adding an abstactor that preprocesses the inputs. It is called *abstract miter* [154]. It is logically equivalent to the one presented in Section 5.3, hereinafter referred to as *concrete miter*. The fundamental property of the abstract miter is that, under some preconditions, it has the very same CSs as the concrete one [154]. From the SMT formula of the abstract

miter, we obtain a pure Boolean model by replacing each abstract stage, i.e., each CSA, with a boolean component over the input and output predicate variables, computed by means of a local AllSMT(EUF) call. Please note that this abstraction is legitimate for our task, since we are not interested in the data exchanged between modules, but just on the conditions that lead to a faulty behavior. The resulting formula is a Boolean formula consisting of fault variables, Boolean input ports of the concretizers, and the Boolean output port of the abstractors. Since we want the resulting formula that models the CSs of the architecture to contain only fault variables, we have to perform an additional processing on the entire formula in order to quantify out Boolean inputs and outputs. This step however can be efficiently performed by using BDD-based projection techniques. The use of predicate abstraction, as presented above, highly improves the performance of our task [154].

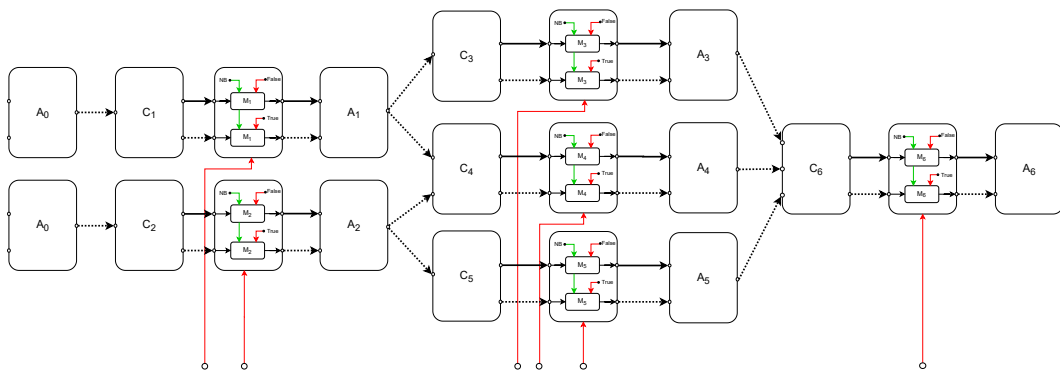


Figure 5.11: Abstract miter

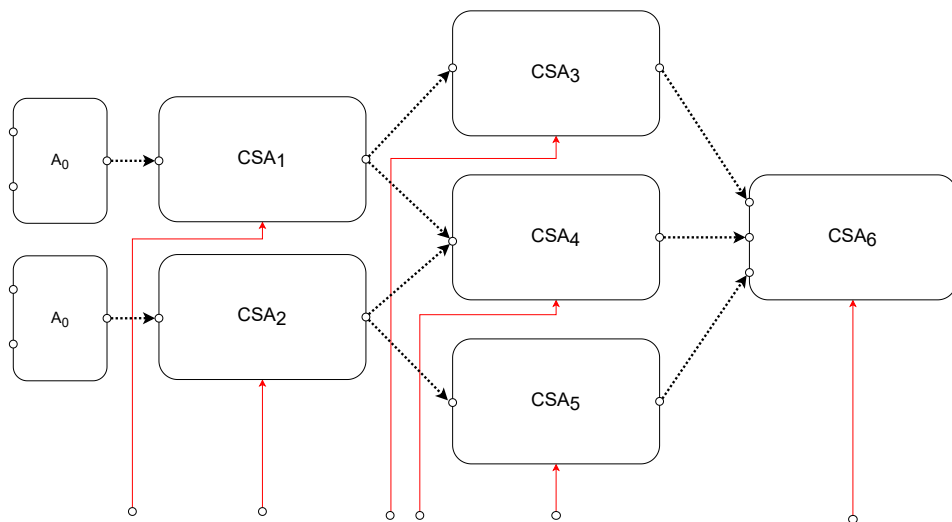


Figure 5.12: Abstract miter is composed by abstract stages

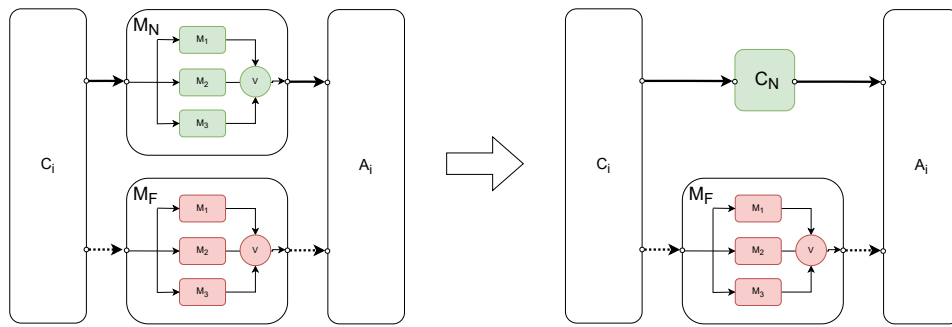


Figure 5.13: Reduced CSA for a TMR example pattern

5.6.2 Reducing the Number of Decision Variables

The performance of the algorithm and the size of the symbolic probability function depend on the size of the OBDD. In order to lower the time needed to perform AllSMT calls on the CSAs and the subsequent quantifier elimination of Boolean input and output ports on the global formula, we can reduce the number of variables by observing that each abstract stage consists of two copies of the same pattern: one for the nominal behavior and another for the faulty behavior. Since in the nominal pattern the fault variables are constrained to be False, i.e., the nominal pattern is ideal, we can substitute it with a nominal component, as illustrated in Figure 5.13 for a TMR. This allows us to halve the number of variables, with a consequent reduction of the computing time.

In addition, the number of variables in the Miter composition can be reduced further by maximizing the sharing of fault variables among patterns allocable to the same basic component that produce different configurations.

Furthermore, one more method to reduce the number of variables is to limit as possible the number of linking constrains. For instance, if the output port out_1 of a component C_1 is connected to the input port in_2 of another component C_2 , rather than of encoding this information by means of the linking constraint $out_1 = in_2$, using two variables, we can share a single port p , which is the output for the first component and at same time the input for the second one, using only one variable.

5.6.3 Management of Uncertain Cases

As stated above, in case of failure, a module produces an arbitrary real value. In a case of bad luck, wrong behavior could produce the right result, i.e., a faulty redundant pattern could provide a correct outcome. This means that

some configurations of fault events at times trigger a TLE, at others do not. To take into account this eventuality, the Boolean formula of a CSA should encode both cases, causing a combinatorial growth in size of the abstracted Boolean formula of each CSA. Let us take a practical example with the TMR. Let assume that modules M_1 and M_2 are faulty (while M_3 and the voter V have nominal behavior), and the CSA receives correct values (i.e., $C_i = True$). If we implement that described above, after executing an AllSMT call on the CSA, the resulting Boolean formula will encode the fact that when two modules of a TMR are faulty, assuming that the voter behaves correctly, the output of the pattern could either be nominal or faulty (i.e., $A_i = True$), having the following form:

$$\begin{aligned} & \dots \vee C_i \wedge F_{M1} \wedge F_{M2} \wedge \neg F_{M3} \wedge \neg F_V \wedge \neg \mathbf{A}_i \vee \\ & C_i \wedge F_{M1} \wedge F_{M2} \neg \wedge F_{M3} \neg \wedge \neg F_V \wedge \mathbf{A}_i \vee \dots \end{aligned}$$

Moreover, another particular case leading to the same phenomenon is possible. According to equation 5.3, if the (non-faulty) voter receives three different inputs, one of these is randomly chosen and provided as output. As in the previous case, the output of the voter can be right (because coming from a non-faulty module) or wrong (because coming from a faulty module). We are interested in finding all the combinations of fault events that lead to a failure of the entire system. For this reason, we should include those cases in the CS computation, even if they could lead to a correct behavior (not triggering the TLE). However, to avoid unwanted behaviors, and at same time reduce the Boolean formula retrieved by ALLSMT calls, we can force the faulty behavior of each sub-component to provide an arbitrary value as long as it is different from the nominal value. Formally speaking, in order to prevent the first uncertain scenario, we have to add to equation 5.8 the following inequality constraint for each module i :

$$beh_{MiF} \neq beh_{MiN} \tag{5.14}$$

And in order to prevent the second uncertain scenario, we have to add the following constraint:

$$o_{M0} \neq o_{M1} \wedge o_{M1} \neq o_{M2} \rightarrow \neg(o_V = o_{M0} \vee o_V = o_{M1} \vee o_V = o_{M2}) \tag{5.15}$$

5.6.4 Caching Mechanism

A redundant pattern is composed by multiple sub-components. For instance, the pattern *TMR_V111* (i.e., a TMR with one voter) has four sub-components: three computing modules and one voter. Instead, the pattern *TMR_V123* (i.e., a TMR with three voters) has six sub-components: three computing modules and three voters. The more complex a pattern is, the higher is the number of possible behaviours described by its CSA formula, and therefore the higher is the number of its models. Furthermore, also the arity of the basic components to which patterns are applied influences the number of variables used to model the connected concretizer. This because each input port of every module of a pattern is modeled by Real SMT variables and the number of inputs of the pattern determines how many variables are needed to model the connected concretizer. For the above reasons, the abstraction (i.e., AllSMT computation) of the behaviour of each pattern is by its nature computationally expensive. It could require a very long time depending on the number of system components, their connections, and their arity. We can save computing time observing that the same type of pattern has a unique behaviour, regardless the basic component it is applied to. A *caching mechanism* can be therefore implemented to drastically reduce the time for the creation of the Boolean formula modeling the combinatorial abstract miter. Once the formula without quantifiers has been computed for a pattern, it is stored and imported every time such pattern is contained in the library of a component (see Figure 5.14). The import from the cache is in general very efficient and it takes less than one second even with the most complex redundant patterns.

Table 5.1 illustrates the comparison of performance when computing the formulae for the first time and when computing the same formulae using the caching mechanism. The system under test is the running example presented in Section 7.

5.7 Work Extensions

We extend the work mentioned above along several directions. First of all, Bozzano et al. [154] study the reliability of a *given* redundant architecture. Instead, we deal with a DSE process in which design alternatives are evaluated to *optimize* the architecture. Our method includes the reliability assessment as part of the evaluation of alternative designs.

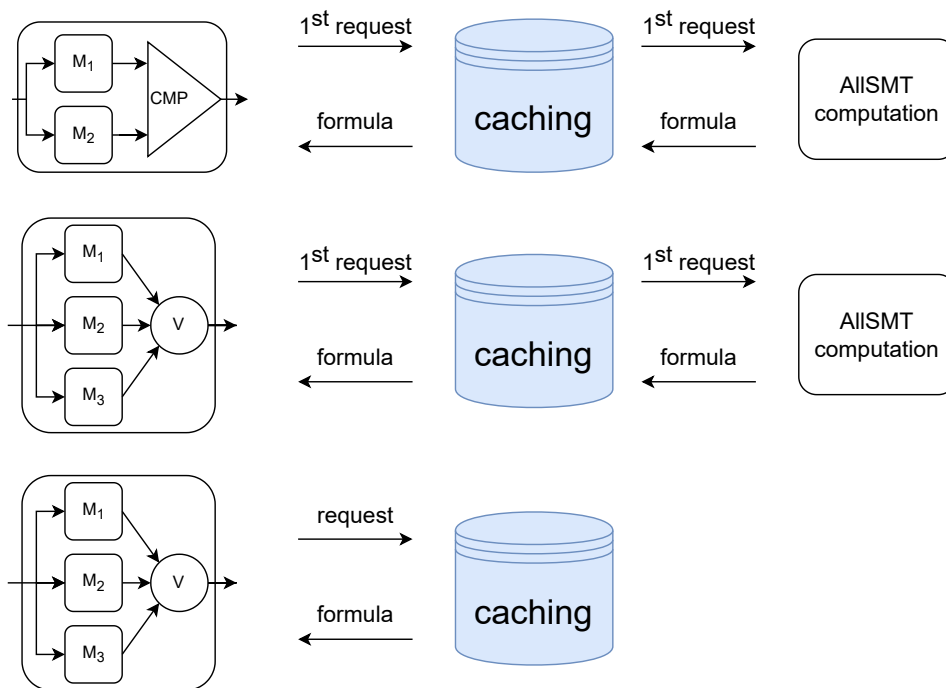


Figure 5.14: Caching improves the performance by storing patterns behavior formulae and accessing them on later requests.

Moreover, we introduce a novel assessment of reliability based on configuration and fault variables, presented in next chapter, that affects the structure of the BDD representing the formula of the deviations from nominal behavior of all valid redundant configurations, pushing the BDD to work in a kind of deductive manner.

In addition, we also consider other non-functional parameters, to find the assignments of redundant design patterns to basic components that optimize (minimize and/or maximize) some objective functions, facing therefore a MOOP.

Furthermore, Bozzano et al. [154] adopted TMR as redundancy architecture, while we build a library of redundancy architectures applicable to the basic components, dealing with the complexities related to their selection and connections.

They implemented the approach by leveraging several existing tools: they used the OCRA language [293] to specify the architecture under analysis, and employed the language SMV [155] and its associated tool XSAP [156] for the symbolic representation of the system. In our case, architecture modeling is performed completely using SMT. Lastly, we also integrate the exact method with a heuristic that can help solve the optimization problem when the sheer

Total Patterns	Caching	Time elapsed [s]	Memory usage [Mb]
12	No	100.964	2.220
	Yes	0.825	2.046
18	No	212.336	2.222
	Yes	1.429	2.202
24	No	580.579	2.272
	Yes	43.535	2.225

Table 5.1: Time and memory performance for the complex system of the running example illustrated in Figure 7.2

size of the design space makes enumerating every design point prohibitive. Besides, we introduce some refinements and smart solutions in every step of our method, in order to further improve the performance.

Chapter 6

Design Space Exploration of Redundant Architectures

In the following we present how to obtain an automated and optimal allocation of redundant component instances. The main challenge in DSE arises from the sheer size of the design space that must be explored.

6.1 DSE Features

There are a number of approaches to DSE. These range from simple brute force approaches to more complex mechanisms mimicking processes such as genetic evolution. Anyway, all these approaches share a common set of features:

- **Design space representation:** a suitable representation of the design space is essential. The representation should be formal, so that it can be subject to automated analysis and exploration techniques.
- **Design space generation:** definition of the complete set of potential and feasible design architectures.
- **Exploration method:** the framework must provide a method for navigating to interesting solutions.

- **Evaluation:** estimation of non-functional proprieties on the basis of a set of parameters.
- **Selection:** pruning the design space from a large pool to a smaller, Pareto-optimal set.
- **Refinement:** further improvement to provide more accurate results.

In the following, we present how these features are designed in our method.

6.1.1 Design Space Representation

An architecture alternative is an assignment of redundancy patterns for all components. Each architectural parameter relevant to us becomes an objective function. Each component of the basic system architecture has a set of assignable patterns (usually assigned on the basis of its data-flow type and/or the kind of faults that the patterns can manage). The architecture is modeled using uninterpreted function to express computing modules, and Boolean logic to express the connections between them. Uninterpreted functions allow us to express the functional properties of the behavior of the components. In addition, being uninterpreted, we can concentrate on the features of the redundant architecture, abstracting from the specific module implementations.

6.1.2 Design Space Generation

One redundant architecture is obtained by choosing one redundant pattern for each component, and thus the full redundant architectures space can be obtained by a full factorial enumeration algorithm. However, some combinations of alternatives are not valid and must be therefore eliminated from the architecture space by means of constraints.

6.1.3 Exploration Method

The goal is to determine the set of all Pareto optimal solutions. This task can be computationally intractable, and, hence, usually it is preferable to approximate the Pareto set as well as possible in a given amount of time. We adopt an exact method: we can take advantage of the existing highly-optimized SMT solvers without having to dive into their intricate implementations, and directly benefiting from future advances in SMT solving. Thus, we can treat

the underlying SMT solver as a black-box, using it to generate models and check validity.

6.1.4 Evaluation

The goal is to find solutions with optimal objective function values or at least improving the quality of the Pareto front approximation. To drive the search and to evaluate the quality of an assignment, we require a so called *fitness function*. To compare the output of multi-objective optimizers we can use dominance relations between sets. An advantage of dominance-based algorithms is that they deal with an archive of solutions rather than a single solution. Therefore, they can return quickly numerous non-dominated solutions to the problem. However, this can also be a drawback since dealing with possibly many solutions can make the exploration of the search space slower in terms of progressing towards high-quality regions of the Pareto front.

6.1.5 Selection

The adopted criterion used to discriminate among solutions in the multi-objective context is the Pareto dominance. By definition, a feasible solution x Pareto-dominates another point x' if the following relation holds:

$$f(x) \leq f(x') \wedge f(x) \neq f(x')$$

in which the relation operators \leq and \neq are defined as follows:

$$\begin{aligned} f(a) \leq f(b) &\iff f_i(a) \leq f_i(b), \forall i \in \{1, 2, \dots, m\} \\ f(a) \neq f(b) &\iff \exists i \in \{1, 2, \dots, m\} : f_i(a) \neq f_i(b) \end{aligned}$$

in which a and b represent two different decision vectors. This dominance is usually expressed as $f(x) \prec f(x')$. Consider an order binary relationship, \preceq , between any two elements in the objective space. Without loss of generality, only minimization is considered. In Table 6.1, we present the relationships between objective vectors and sets of objective vectors used in this paper.

6.1.6 Refinement

At each stage of our method, different refinement solutions will be investigated in order to improve speed and efficiency.

Table 6.1: Relations

Relation	Objective vectors	Meaning	Sets of objective vectors	Meaning
Dominance	$f(a) \prec f(b)$	$\exists i, f_i(a) < f_i(b)$ and $\forall j \neq i, f_j(a) \leq f_j(b)$	$A \prec B$	$\forall f(b) \in B, \exists f(a) \in A$ <i>s.t.</i> $f(a) \prec f(b)$
Weakly dominates	$f(a) \preceq f(b)$	$\forall i, f_i(a) \leq f_i(b)$	$A \preceq B$	$\forall f(b) \in B, \exists f(a) \in A$ <i>s.t.</i> $f(a) \preceq f(b)$
Incomparable	$f(a) \parallel f(b)$	$f(a) \not\preceq f(b)$ and $f(b) \not\preceq f(a)$	$A \parallel B$	$A \not\preceq B$ and $B \not\preceq A$
Non-dominated by	$f(a) \not\preceq f(b)$	$f(a) \preceq f(b)$ or $f(a) \parallel f(b)$	$A \not\preceq B$	$A \preceq B$ or $B \parallel A$

6.2 Constraint Solving Approach

Although the DSE is considerably less complicated than the overall problem that we need to solve in the development of ES (i.e., the deployment on a specific target), it is not easy to find a good and fast algorithm that we could use to solve efficiently any instance of the redundant architecture problem. Because of the complex structure of the problem and the heterogeneous nature of its instances, we have decided that the best course of action would be to employ a flexible and adjustable method. This is how we came up with the idea of using the constraint solving approach. Such approach has some key advantages. Firstly, it is applicable to all the instances of the redundant architecture synthesis problem and is reasonably efficient in practice. In addition, as we want to find optimal solutions, constraint solving supports optimization intrinsically. Furthermore, there are numerous state of the art solvers available on the market, giving us the chance to choose among them when implementing this solution in practice. Moreover, further adaptation of the solving process is possible. In fact, many solvers offer methods of adapting their search strategy in function of the given problem, or by using high-level knowledge about the specific problem's structure. More in general, the constraint solving approach is flexible with regard to the changes in the underlying problem.

6.2.1 Formalization of Constraints

In the following we illustrate how system constraints can be formalized. For example, product cost is a very important factor in industrial projects. It is thus relevant to define a constraint to limit component costs, without violating any other constraints. It is possible to calculate the total cost of the system as the sum of the single costs of the patterns used in the current redundancy

allocation. Subsequently, we restrict the maximum cost of the system as follows:

$$total_cost \leq max_cost$$

where

$$total_cost = \sum_i cost(pattern_i)$$

Similarly, we can constrain the whole *power consumption* of the used hardware, or apply a specific constraint. If we define the consumption of a node n as $n.energy$, the total consumption of the system can be formalized with the following equations:

$$total_energy_cons \leq max_energy$$

where:

$$total_energy_cons = \sum_n n.energy$$

To reduce the complexity of the hardware architecture we can impose constraints to the whole system size or the number of devices. We can also limit the *number of available identical units* from each redundant pattern type.

$$total_used_pattern_i \leq max_number_pattern_i$$

where:

$$total_used_pattern_i = \sum_i i.used$$

We can also formalize constraints on the *deployment* (at logical level). For instance, some applications may require special computation hardware, access to sensors and actuators, communication interfaces, etc. Thus, the designer should be able to specify the required and available skills, and allocate particular functions to particular devices, or constraints the deploying of some functions onto multiple resources to achieve a parallel execution. Similarly, he may want to couple some functions, so that they are executing within the same device, or within the same resource. For instance, suppose there is a function that includes two tasks that communicate with each other, and we want to deploy them on two different nodes. They have to communicate over a bus b , which also have a fault model. If the task t_1 sends messages to t_2 , and those

tasks are mapped on different nodes, the constraint can be formalized as:

$$t_1.safety \geq t_2.safety$$

where *safety* could be another objective about safety requirements that addresses the severity of failures. To provide a safety assessment method at abstract level of design patterns, we can define a safety metric based on the computation of the relative safety improvement achieved when using the design patterns under consideration, basing for example our calculations on the safety recommendations of the IEC 61508 standard (the international standard for the functional safety of electrical, electronic, and programmable electronic equipment), which is the most general standard that can be applied to any safety-related system. We could also provide a method to derive general safety recommendations for safety integrity levels at the abstract level of design patterns, computing.

More in general, using this formalization, we can impose any kind of constraints.

6.3 Problem Encoding

System reliability enhancement usually comes with a higher cost, size and power consumption, and lower performance. For the design of a reliable system, it is desired to not only maximize reliability but also minimize cost, size and power consumption, and maximize performance. Design problem can be therefore formulated as a MOOP. The aim is to find the solution vector of the redundancy allocation for the system, or vector of n design variables

$$X = (x_1, x_2, \dots, x_n)^T$$

that optimizes a vector of p (conflicting) objective functions

$$F(X) = (f_1(X), f_2(X), \dots, f_p(X))$$

over the *feasible region* of design space X formed by the vector of m constraint functions

$$G(X) = (g_1(X), g_2(X), \dots, g_m(X)).$$

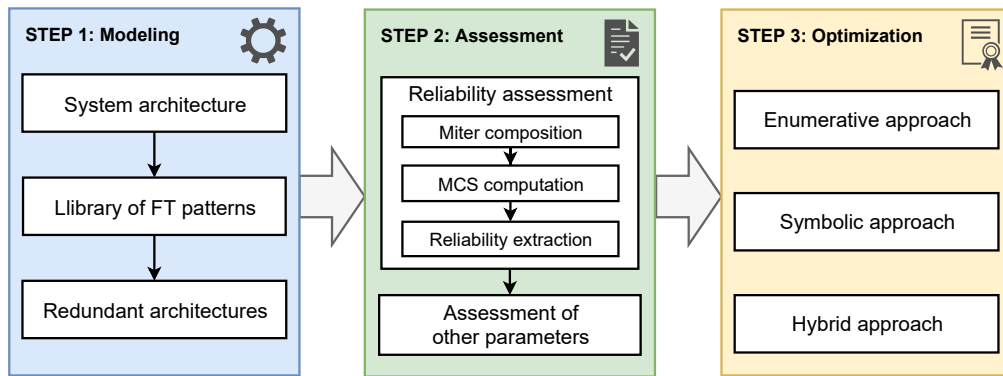


Figure 6.1: The proposed method consists of three main phases: modeling of redundant system, assessment of non-functional parameters, and optimization.

The problem can be therefore loosely written as follows.

$$\min F(X)$$

$$s.t. G(X) \leq 0$$

where:

$$X \in \mathbb{R}^n$$

$$F : X \in \mathbb{R}^n \mapsto \mathbb{R}^P$$

$$G \in \mathbb{R}^n \mapsto \mathbb{R}^m$$

In the following, we discuss the problem from an optimization perspective where redundant system synthesis can be considered a COP. We propose a *multi-objectives DSE* process that *automatically* selects the appropriate set of FT techniques (at logical level) to be applied to the basic (non-redundant) system to obtain a redundant one, optimizing simultaneously a collection of objective functions

The steps of our method are outlined in Figure 6.1.

The automatic DSE proposed is heavily based on the reliability assessment of redundant architecture candidates, which we have discussed in Chapter 5. To evaluate the probability of failure of a redundant architecture, we are going to express the system reliability in terms of the reliability of individual components. We are going to express the constraints as SMT formulae. When solving an SMT COP, the standard optimization procedure is called each time the SAT engine reaches a feasible assignment. It is vital to the efficiency of the

algorithm to reduce the number of calls to classical optimization procedures. If it is possible to obtain more information other than the optimal solution in the optimization subroutine, the whole searching process might benefit a lot. That is why, on large optimization problems, computationally too hard to explore the whole search space, incomplete search algorithms often find a solution much more quickly than complete algorithms. The incomplete search algorithms search for a solution heuristically without passing by the whole search space. Neighborhood (or local) search algorithms are the most successful class of approximate algorithms where at each iteration an improving solution is found by searching the "neighborhood" of the current solution. The weakness of incomplete search algorithms is that they are not able to determine whether a solution exists or not, and they have no guarantee to find a solution if it exists. Anyway, our case is of practical application and input formulae are expected to be satisfiable, making them well-suited for complete search algorithms. Despite known successes, exact methods have also some disadvantages. First of all, the computational time increases strongly with the instance size. In addition, the memory consumption of exact algorithms can be very large and lead to the early abortion of a program.

The steps of our method are reported in Algorithm 1 and detailed in the following according to the flow illustrated in Figure 6.2.

Algorithm 1 DSE general framework

- 1: Modeling the system architecture
 - 2: Construction of a library of redundant design patterns
 - 3: Modeling the redundant architectures
 - 4: Generation of all possible redundant configurations
 - 5: Modeling the Miter
 - 6: Assessment of reliability and other non-functional requirements
 - 7: Perform Optimization
-

6.3.1 Modeling the System Architecture

A high-level system architecture model is our starting point. We describe the basic (non-redundant) system via data-flow modeling, which examines how data moves around the system, focusing on how things connect. The simplest way to implement that is to specify for each node n a list of neighbors, i.e. nodes connected to n . This means that we are going to model the architecture in terms of connections between its components. The basic system is therefore

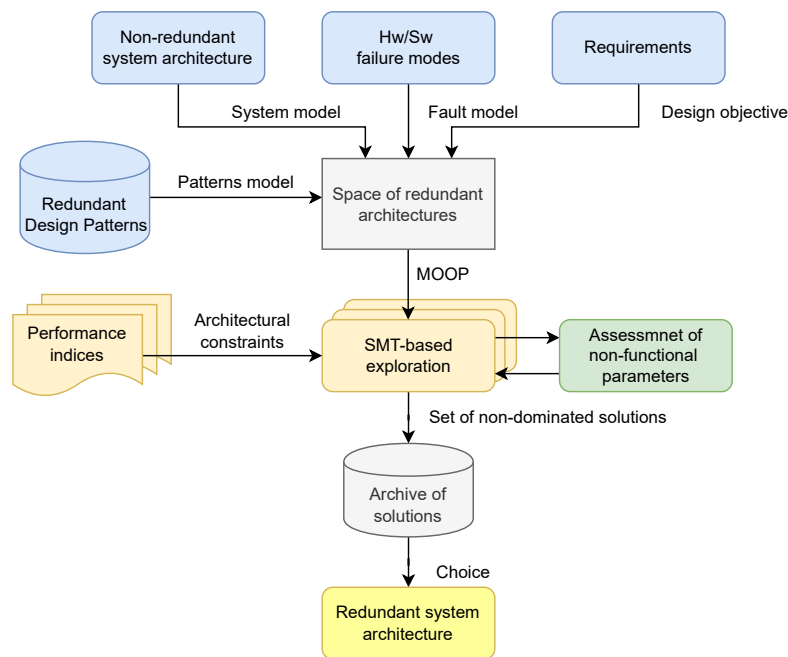


Figure 6.2: Multi-objective DSE flow

represented by DAG in which nodes represent the components of the architecture (denoted by C_i) and edges describe how components are connected.

6.3.2 Construction of a Library of Redundant Patterns

The concept of design patterns is a universal approach to describe common solutions to widely recurring design problems. A design pattern is an abstract representation for how to solve a general design problem which occurs over and over in many applications. To construct the library of redundant design patterns, widely used and proven solutions in the field of ES have to be collected from literature. These solutions should be generalized establishing a high-level and abstract representation which can be used to evaluate the impact of the collected solutions when applied to the basic system architecture, independently from a specific application or a specific implementation.

- **Comparator (CMP)**. Two components perform the same task in parallel and their results are compared by a comparator. A fault can be detected if the results disagree. In this case, the switch can shut down the system or switch to a fail-safe state. CMP will work correctly as long as the two components, the switch, and the comparator have no faults. Assuming that the faults of the two computing elements and of the voter

are independent, and since the two components are identical (i.e. same failure rate), the probability of failure is symbolically expressed as follows:

$$P_{failure} = F_C + (1 - F_C)(F_M^2)$$

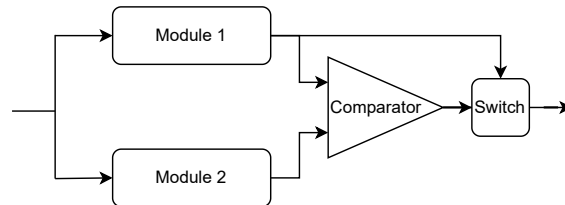


Figure 6.3: Comparator design pattern

Since there are two identical modules, the recurring cost, power dissipation, and size will be increased by 200% comparing to the basic module (ignoring the comparator).

- **Duplex (DPX).** It is a pattern used to increase the reliability and safety of the system by providing a replication of the same module to deal with the random faults. It is based on the assumption that it is highly unlikely for two identical components to suffer a random fault simultaneously. Generally, it consists of two identical modules, a primary and a secondary, and a fault detection unit that monitors the primary module and switches to the secondary module when a fault occurs in the primary. Without faults, the two components give the same result, so the primary component is used to accomplish the required task, but when there is a fault, a fault-detector detects it and generate an instruction to switch to the secondary component. DPX will continue to work correctly as long as one of the two channels has no fault. Assuming that the faults of the two computing elements and of the fault detector are independent, and since the two components are identical (i.e. same failure rate), assuming that the probability of failure for a module M is F_M , and for the detector is F_{fd} , the probability of failure is symbolically expressed as follows:

$$P_{failure} = F_{fd} + (1 - F_{fd})(F_M)(1 - F_M)$$

Cost, power dissipation, and size will be increased by 200% comparing to the basic module (ignoring the fault detection unit). In case of heterogeneous modules, i.e., employing two independent different modules,

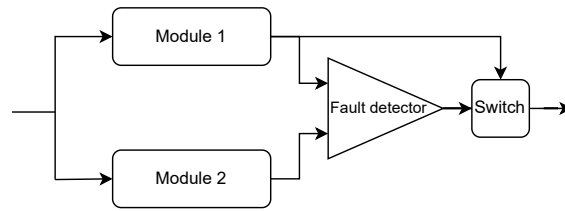
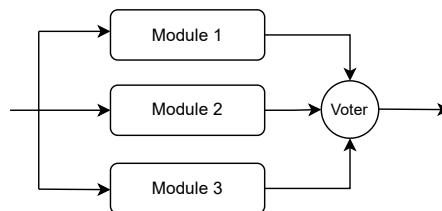


Figure 6.4: Duplex design pattern

development cost that includes two independent designs and two development teams to provide independence of systematic faults should be considered. Please note: with our formalism and fault model considered, CMP and DPX will collapse in the same model.

- Triple Modular Redundancy (TMR).** It consists of three identical modules that operate in parallel to detect random faults, in order to enhance reliability and safety. The modules produce three results that are compared using a voting system to produce a common result as long as two channels or more have the same result. The redundant system will not fail if the voting circuit does not fail and if none of the three modules fails, or if exactly one of the three modules fails. It is assumed that the failures of the three modules are independent. Since the two events are mutually exclusive, the reliability of the redundant system is equal to the sum of the probabilities of these two events. Hence, assuming that the faults of the three computing elements and of the voter are independent, and since the three components are identical (i.e. same failure rate), assuming that the probability of failure for a module M is F_M , and for a voter V is F_V , the probability of failure is symbolically expressed as follows:

$$P_{failure} = F_V + (1 - F_V)(3F_M^2 - 2F_M^3)$$

Figure 6.5: TMR design pattern (TMR_V111)

Since there are three identical modules, we can consider an increase of cost, power dissipation, and size of 300% comparing to the basic module. TMR patterns may have up to three voters, and different connections between them, resulting in various combinations. Figure 6.6 shows various TMR configurations, with one, two, and three voters. In these cases, cost, power dissipation, and size vary accordingly.

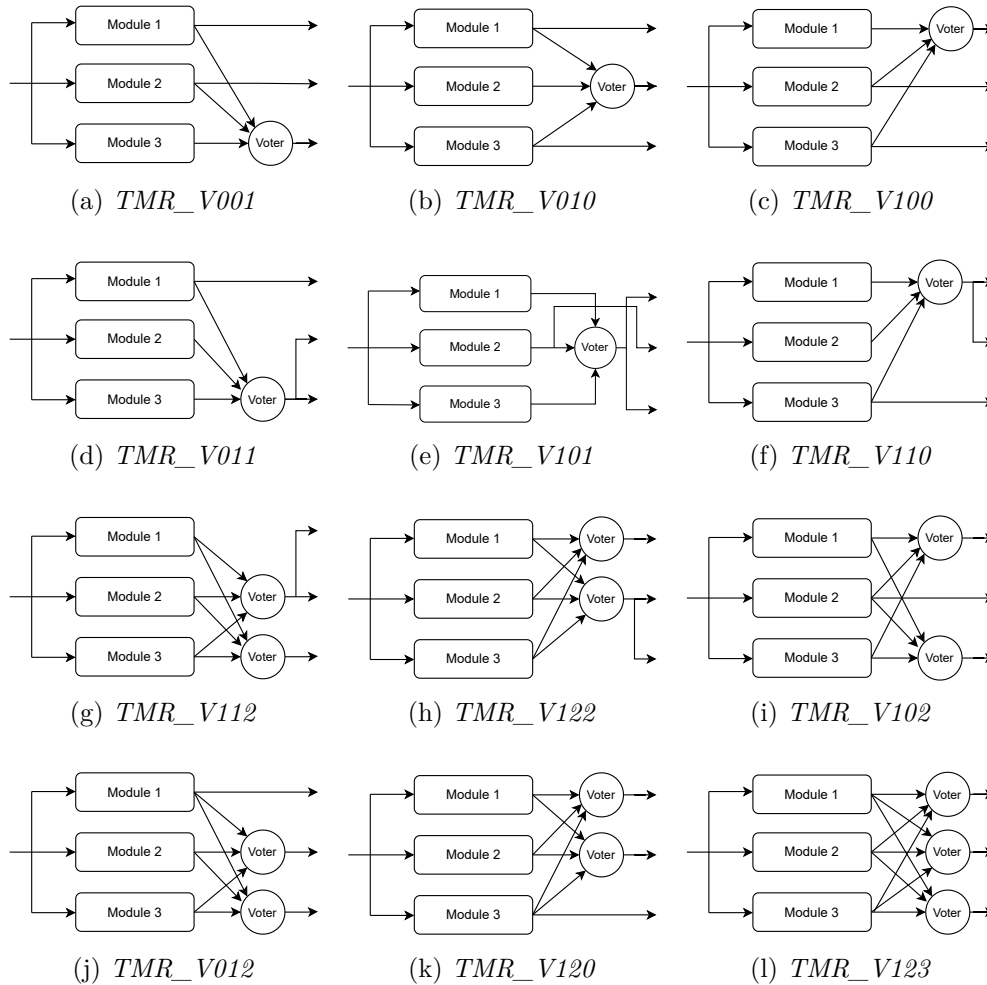


Figure 6.6: Different TMR configurations

- **M out of N (M-oo-N)**. It consists of N identical modules which operate in parallel to mask random faults, and to enhance system safety and reliability. It requires that at least M components succeed out of the total N parallel modules for the system to succeed. The redundant system will continue to work correctly as long as at least M modules have no fault, and the voting circuit does not fail. For example, assuming the voting

circuit does not fail, a 3-o-o-5 will work correctly as long as three or more components have no faults.

$$P_{failure} = F_V + (1 - F_V) \left(\sum_{i=M}^N \binom{N}{i} (F_M)^i (1 - F_M)^{N-i} \right)$$

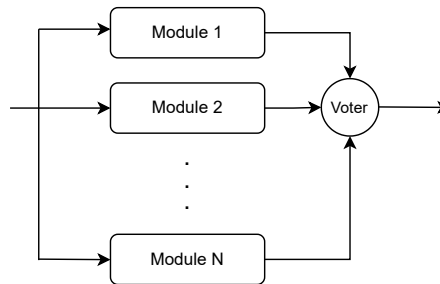


Figure 6.7: M-o-o-N design pattern

The increase of cost, power dissipation, and size can be considered approximately $N \times 100\%$ comparing to the basic system.

- **Sparing (SPR).** If a fault is detected by a built-in error detection unit in the active component, a spare component takes over. The redundant system can tolerate $N-1$ faults as long as the switching circuit does not fail.

$$P_{failure} = (F_C + (1 - F_C)(F_M))^N$$

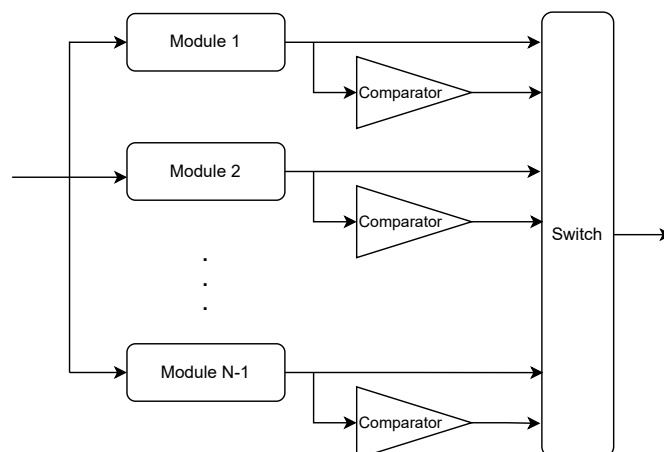


Figure 6.8: Sparing design pattern

Ignoring the comparators and the switch, the increase of cost, power dissipation, and size is $N \times 100\%$ comparing to the basic system. Please

note: with our formalism and fault model considered, M-oo-N and SPR will collapse in the same model.

Of course, the library can be considered as extensible: we have the ability to add new patterns.

6.3.3 Fault Model

We can augment the original architecture model with fault information to characterize anomalous conditions, producing the architecture fault model. As we have done in Section 5.2, the occurrence of faults is modeled with the introduction of variables that enable the components to have internal failures. A failure leads a component to behave incorrectly and produce an arbitrary output. Failure probabilities are given and can be considered constant during the life-cycle of each component, or expressed as function of time. Components are extended by defining two separate behaviors: nominal and faulty, both represented as *uninterpreted functions*. Each extended component receives as parameters: the input representing the input values of the computation (of type real), a Boolean parameter *can_fail* that enables the component to have internal failures, and the nominal behavior of the computation (see Figure 6.9). The outputs of each pair of nominal and faulty components are provided to a multiplexer, which selects the proper signal according to the fault event. Denoting by F_C the probability of fault of a given component, the formal model that describes the setting shown in Figure 6.9 is defined using SMT with the following formula.

$$\neg F_C \implies output = nominal_behavior(input). \quad (6.1)$$

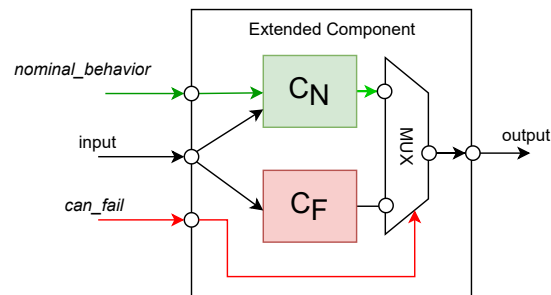


Figure 6.9: An example of extended component

Equation 6.1 states that if no failure occurs, the output is the nominal one, otherwise the output is arbitrary (hence, it is not specified).

6.3.4 Modeling the Redundant Architecture

We can use the same approach to model the redundant components: Figure 6.10 illustrates this modeling technique applied to a TMR.

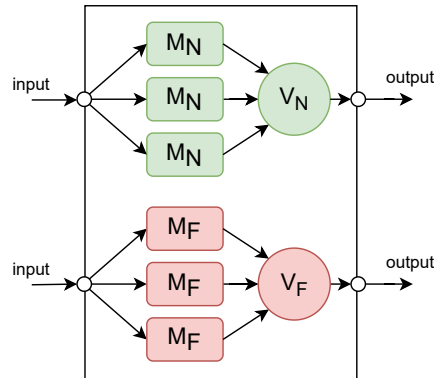


Figure 6.10: An example of extended component

In addition, since an ideal redundant pattern is made up of sub-components that never fail, the output of such pattern must be equal to the output of the nominal version of the basic component to which it is allocated. For this reason, we can apply the refinement presented in Section 5.6.1, and reduce each stage by substituting the nominal pattern with a nominal component, halving in this way the number of faulty variables, obtaining the setting illustrated in Figure 6.11.

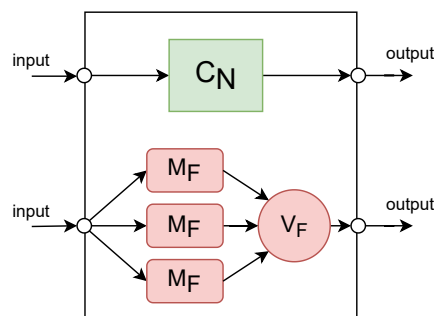


Figure 6.11: Extended TMR

The computing modules can be represented as uninterpreted functions, while voters and comparators as logical formulae (because they have well defined implementations). Using a formal notation, the single redundant components of a redundant architecture can be modeled as combinatorial elements

equipped with sets of Boolean fault variables (that determine the behavior when one or more faults occur), and can therefore be defined as follows.

Definition 3 (Redundant component). *A redundant component C_{iR} is a tuple $\langle \vec{I}, \vec{O}, \vec{F}, \pi \rangle$ where:*

- \vec{I} is the vector of inputs
- \vec{O} is the vector of outputs
- \vec{F} is the set of faults events
- $\pi(\vec{I}, \vec{O}, \vec{F})$ is an SMT formula

Leveraging this approach, we can model via SMT an entire redundant architecture, specifying the connections by imposing that the outputs of a component are equal to the inputs of the subsequent one, using linking constraints (see Definition 2).

6.3.5 Generation of All Redundant Configurations.

Each component C_i of the basic system architecture has a set of assignable patterns P_j form the library of FT patterns, designated for example on the basis of its data-flow type and/or the kind of faults that the patterns can manage. We can define a variable to keep track of all valid allocations (C_i, P_j) .

Definition 4 (Configuration variable). *A configuration variable $cfg_x = (C_i, P_j)$ is a Boolean variable that defines the mapping between a component C_i of the basic system and a redundant pattern P_j of the library, such that $(C_i, P_j) = 1$ iff P_j is a valid pattern for C_i .*

Each configuration variable translates into a *building block* of the redundant architecture. Each block is defined as in Definition 3, i.e., it is characterized by one or more inputs, one or more outputs, some faulty variables, and a behaviour represented as a set of SMT constraints describing the input-output relationship. Configuration variables are arranged in a configuration vector \vec{cfg}_i specifying all the possible allocations of candidate patterns to the basic component C_i , in order to define a redundant architecture. The set of configuration variables for an architecture is obtained by choosing a valid pattern assignment for each basic component. Hence, the redundant patterns for each

component can be *binary encoded*, and the length of the configuration vector \vec{cfg}_i of a component C_i depends on the number of valid redundant patterns lib_{C_i} allocable to C_i , which is computed as follows:

$$\text{len}(\vec{cfg}_i) = \lceil \log_2(\text{len}(lib_{C_i})) \rceil.$$

Note that some values encoded by the configuration vector may not be associated to any pattern. Each configuration vector \vec{cfg}_i should be therefore constrained in order to encode only meaningful values. Assuming that \vec{cfg}_i is an SMT bit-vector variable (in the theory \mathcal{BV} of bit vectors), we must constraint its values using *configuration constraints*, defined as follows.

Definition 5 (Configuration constraint). *Given a configuration variable $cfg_x = (C_i, P_j)$, a configuration constraint is defined as follows:*

$$\vec{cfg}_i < \text{len}(lib_{C_i}).$$

Configuration constraints are needed in order to prevent the modeled architecture from assuming undefined behaviors. The patterns available for each component could also be encoded using *one-hot encoding*, i.e., a group of bits among which the legal combinations of values are only those with a single high (1) bit and all the others low (0) [296], but in this case it would be necessary to add the mutex constraint on the configurations that cannot occur, in order to guarantee the hypothesis of monotonicity.

A *redundant architecture* corresponds to an assignment of redundancy patterns for all components. Since each component C_i of the basic system architecture has a finite set of assignable patterns P_j , the number of all possible redundant alternatives is combinatorial, and depends on the number of basic components composing the architecture and on the number of available redundant patterns for each component. This number can be computed as follows:

$$\text{Number_of_alternative_designs} = \prod_{i=1}^n \text{len}(lib_{C_i}) \quad (6.2)$$

Two redundant configurations differ in the way the valid patterns allocated are connected. By combining all valid allocations, we obtain the set of the redundant architectures. Thus, all the redundant architectures are obtained by assigning in turn one (valid) pattern to each component, i.e:

$$\sum_{i=1}^N \text{ite}((C, P_i), 1, 0) = 1$$

Since each redundant pattern has a specific number of input and output ports, two redundant components may be incompatible, i.e., the output ports of the first does not match with the inputs of the second one. To deal with this occurrence, we inhibit some configuration compositions, by introducing what we call *compatibility constraints*.

Definition 6 (Compatibility constraint). *A compatibility constraint is a specialized constraint that inhibits the connections between two redundant components if the number of outputs of the former is not equal to the number of the inputs of the latter.*

As a running example, Figure 6.12 illustrates a system of three components, namely $\{C_1, C_2, C_3\}$, connected in series, and a library of three redundant design patterns providing a total of seven instantiations (the same pattern can be implemented in different ways, depending on the internal sub-components), namely $\{P_1, \dots, P_7\}$, supposing that $\{P_1, P_2, P_3\}$ are valid patterns for C_1 , $\{P_4, P_5\}$ are valid for C_2 , and $\{P_6, P_7\}$ are valid for C_3 . Suppose also that the library is composed as in Table 6.2. We name lib_{C_i} the part of library that includes the patterns allocable to component C_i . We assume that the computing elements have different failure probabilities. Specifically, probability of failure of module M_i is F_i . as illustrated in Figure 6.13. On the basis of the previous assumptions, Figure 6.14 illustrates an example of fault variables involved depending on the configuration of the patterns selected. Note: the number of variables can be reduced as proposed in Section 5.6.2.

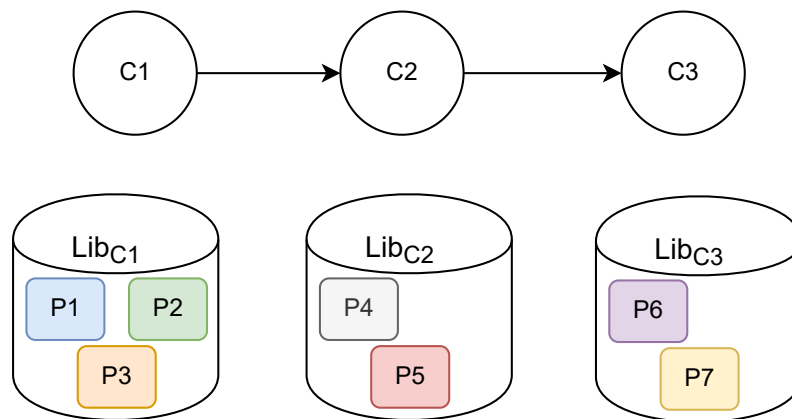


Figure 6.12: Example of basic (non-redundant) architecture composed of three components connected in series and a library of seven redundant design patterns.

Table 6.2: Library of patterns for example system in Figure 6.12

Component	Pattern	Type
C_1	P_1	DPX
	P_2	TMR with 1 Voter (TMR_V111)
	P_3	TMR with 3 Voters (TMR_V123)
C_2	P_4	DPX
	P_5	TMR with 1 Voter (TMR_V111)
C_3	P_6	DPX
	P_7	TMR with 1 Voter (TMR_V111)

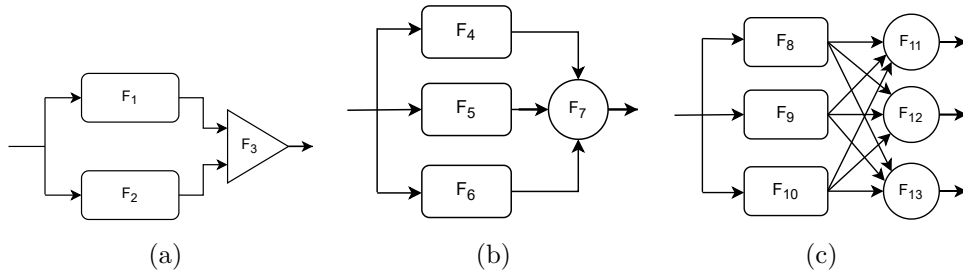
Figure 6.13: Fault atoms for patterns P_1 (a), P_2 (b), and P_3 (c).

Figure 6.15 illustrates how to combine valid allocations in order to obtain the set of redundant architectures for our running example. Combinations of building blocks that generate the redundant architectures can all be modeled in a single formula specifying the linking constraints, according to the configuration.

$$\left((C_1, P_1)=1 \rightarrow \text{block}_{11}.\text{out}=\text{block}_{24}.\text{in} \wedge \text{block}_{11}.\text{out}=\text{block}_{25}.\text{in} \right) \wedge$$

$$\left((C_1, P_2)=1 \rightarrow \text{block}_{12}.\text{out}=\text{block}_{24}.\text{in} \wedge \text{block}_{12}.\text{out}=\text{block}_{25}.\text{in} \right) \wedge$$

$$\left((C_1, P_3)=1 \rightarrow \text{block}_{13}.\text{out}=\text{block}_{24}.\text{in} \wedge \text{block}_{13}.\text{out}=\text{block}_{25}.\text{in} \right) \wedge$$

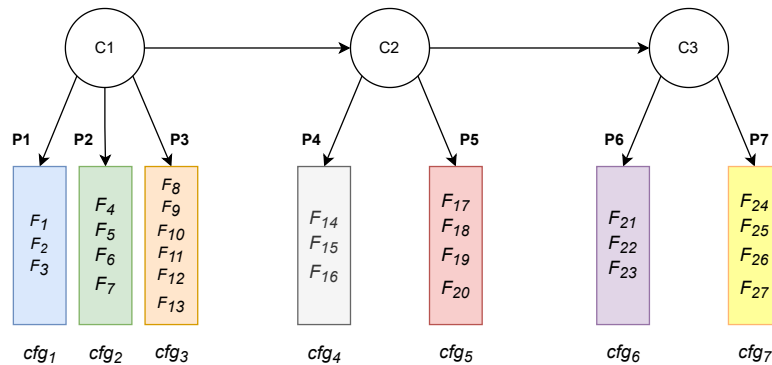


Figure 6.14: Configuration and fault variables

Table 6.3: Binary encoding of (C_i, P_j) allocations

cfg_1	C_1-1	C_1-0
$(C_1 - P_1)$	0	0
$(C_1 - P_2)$	0	1
$(C_1 - P_3)$	1	0

cfg_2	C_1-1	C_2-0
$(C_2 - P_4)$		0
$(C_2 - P_5)$		1

cfg_3	C_1-1	C_3-0
$(C_3 - P_6)$		0
$(C_3 - P_7)$		1

$$\left((C_2, P_4)=1 \rightarrow \text{block}_{24}.\text{out}=\text{block}_{36}.\text{in} \wedge \text{block}_{24}.\text{out}=\text{block}_{37}.\text{in} \right) \wedge \\ \left((C_2, P_5)=1 \rightarrow \text{block}_{25}.\text{out}=\text{block}_{36}.\text{in} \wedge \text{block}_{24}.\text{out}=\text{block}_{37}.\text{in} \right)$$

I.e., for each configuration and for each block_{ij} connected to $\text{block}_{i'j'}$ holds that:

$$\bigwedge_{i=1}^N ((C_i, P_j) \implies (\text{block}_{ij}.\text{out} = \text{block}_{i'j'}.\text{in})), \forall j : j \in \text{Lib}_{C_i}, \quad (6.3)$$

Note that the formulae involved are composed by *configuration variables* cfg_i that determine selected patterns, and *fault variables* F_i that depend on configuration, and indicate whether a certain component C_i is faulty or not.

Figures 6.16a to 6.16l show the redundant architecture alternatives. Table 6.3 reports the binary encoding of the configurations. The first component, namely C_1 , has three available patterns. It requires therefore a configuration vector of length 2. Components C_2 and C_3 have two available patterns each, hence they require a configuration vector of length 1. The failure of each component leads to a TLE.

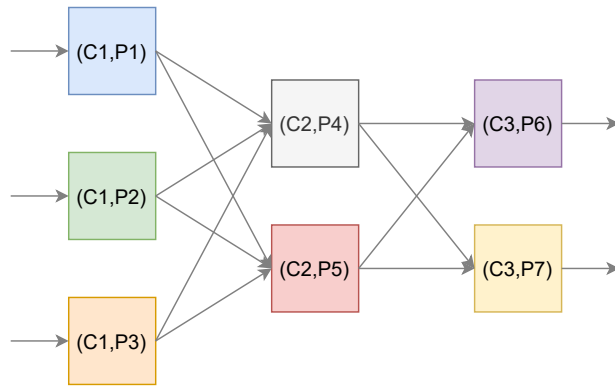


Figure 6.15: Set of redundant architectures for example system

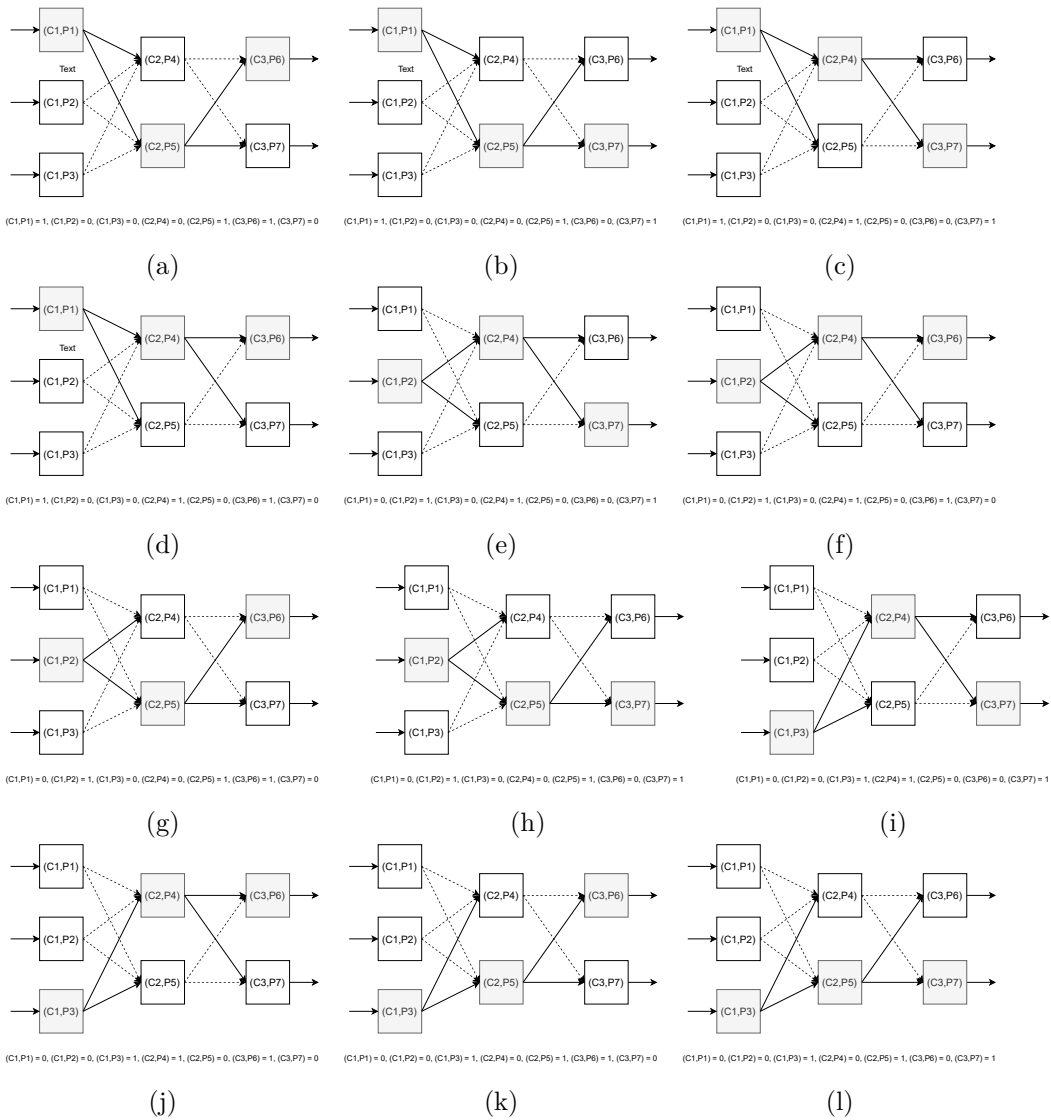


Figure 6.16: Redundant alternatives for system in Figure 6.12

6.3.6 Modeling the Miter

As seen in Section 5.3, with a Miter composition we can detect deviations of the system under analysis from its nominal behavior, i.e. the TLE. Figure 6.17 shows the Miter composition for the alternative architecture of Figure 6.16a, Figure 6.18 shows the stage-based Miter composition for the same architecture, Figure 6.19 shows the respective abstract Miter composition.

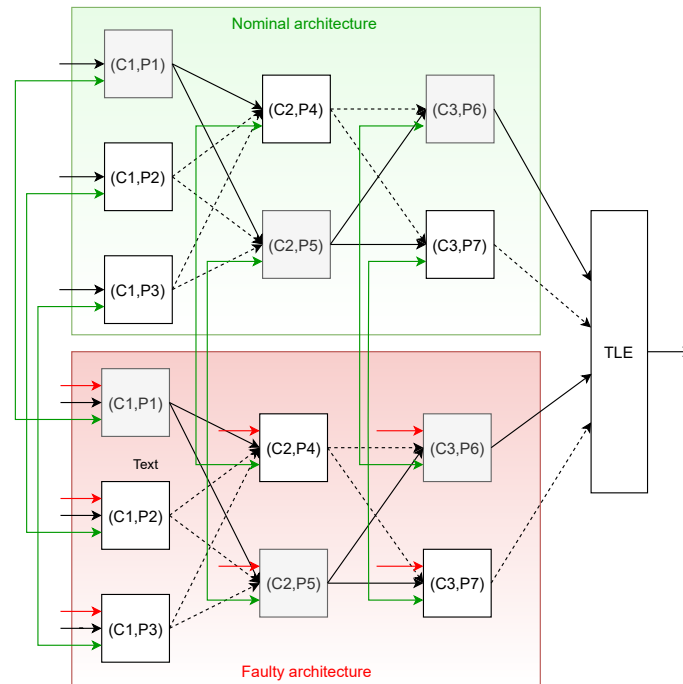


Figure 6.17: Miter composition for architecture of Figure 6.16a

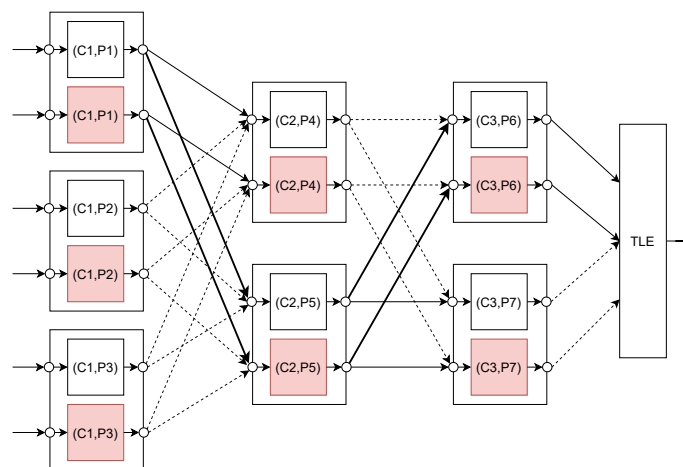


Figure 6.18: Stage-based Miter for architecture of Figure 6.16a

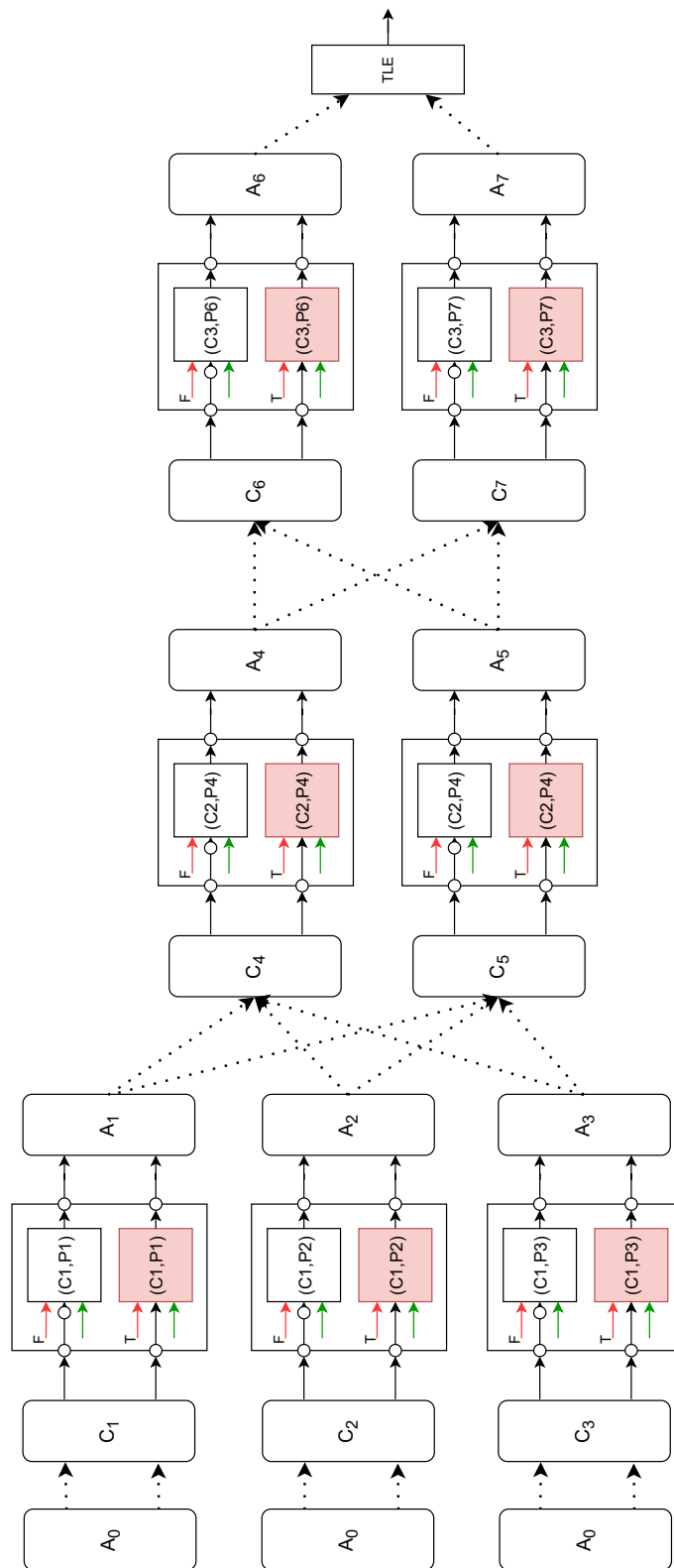


Figure 6.19: Abstract Miter composition for architecture of Figure 6.16a

6.3.7 Reliability assessment

To produce the reliability function, we follow the same steps presented by Bozzano et al. [154]:

- STEP 1: we model the modules and the connections between them as specified above, using uninterpreted functions that allow us to abstract from the specific module implementations, focusing on the features of the redundant architecture.
- STEP 2: We model the abstract Miter described above.
- STEP 3: We construct the set of all fault configurations that are sufficient for the architecture under analysis to fail in order to calculate the set of CSs.
- STEP 4: We generate the Reliability Function by traversing the BDD-based representation of the set of CSs.

In Chapter 5 we presented the method to extract the reliability formula of a given redundant architecture. In the following, we extend that method and present a fully automated approach to the assessment of reliability and other non-functional parameters of *families* of complex redundant architectures, and therefore support the DSE. A key design choice in our approach is how to define the system reliability. Assuming that there is no dependence between the components, we can define the formula for the reliability of the system as a function of that of the individual components. Under this assumption, we could provide the solver with the individual reliabilities (in terms of failure probability) and the global reliability formula of the system, relying on the solver to find the solutions. Please note that with non-homogeneous patterns (for example a TMR with 1 input and 3 outputs) the hypothesis of independence of the components would fail. As alternative, we can make semi-symbolic choices, acting on the components: one could treat some components symbolically, while the others would be made explicit. For instance, if we have 2^4 possible combinations, we could deal with a single problem with four decision variables, or with four problems each of these with only two symbolic possible choices at run-time, or with sixteen different problems, each one corresponding to a possible redundant configuration. Or more in general, we can build a single symbolic representation of the problem, leaving the choice at run-time, leveraging symbolic methods to reduce the design space that needs to be explored.

To sum up, either the system is parameterized and the parameters are made explicit before the search (it corresponds to doing no search), or one of the parameters is fixed and the remaining ones are enumerated, or a single symbolic representation of the problem is used. Once we have the symbolic formula, we have the chance to make all the available choices. All the above solutions have advantages and disadvantages. The problem in which parameters are instantiated is easy to be implemented, and highly parallelizable: for example, by setting 10 parameters, 1024 problems can be generated to be assigned to 1024 different CPUs. The drawback is that we have to analyze the reliability individually on a combinatorial number of possible configurations. With the symbolic approach, the reliability is not calculated but it is represented (in our case using a BDD with an SMT formula), and then can be optimized (with OMT), but it can potentially lead to an exponential number of models.

Explicit Representation

The simplest approach to extract the reliability function consists in considering all possible design alternatives in separate way, and compute the reliability individually, as described in Chapter 5. Since each alternative redundant architecture is composed by a different combination of redundant patterns from the library, we have to compose a new Miter for each alternative. Given the failure probability of each sub-component, we can obtain the value of the overall system failure probability, and easily map each configuration to the corresponding reliability value. Hence, with this method the reliability function is represented as a sequence of implications:

$$configuration \rightarrow reliability$$

Consider for example the first design alternative of our running example, illustrated in Figure 6.20. We can compute its reliability as follows:

$$cfg_1 = [0, 0] \wedge cfg_2 = [1] \wedge cfg_3 = [0] \rightarrow R_{alt1} = 0.93$$

As stated above, the overall reliability function can be expressed as a conjunction of implications. Iterating the procedure for all alternative architectures illustrated in Figure 6.16, we can extract the reliability of the redundant system, which has the following structure.

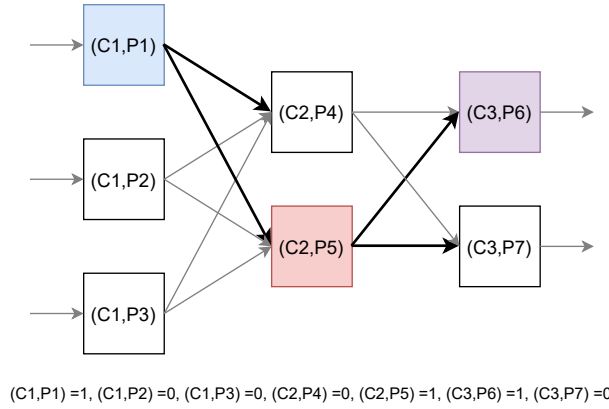


Figure 6.20: Redundant architecture alternative 1 of running example

$$\begin{aligned}
 c f g_1 &= [0, 0] \wedge c f g_2 = [1] \wedge c f g_3 = [0] \rightarrow R_{alt1} = 0.93 \wedge \\
 c f g_1 &= [0, 0] \wedge c f g_2 = [1] \wedge c f g_3 = [1] \rightarrow R_{alt2} = 0.97 \wedge \\
 c f g_1 &= [0, 0] \wedge c f g_2 = [0] \wedge c f g_3 = [1] \rightarrow R_{alt3} = 0.96 \wedge \\
 &\dots
 \end{aligned} \tag{6.4}$$

As stated above, the reliability of each configuration can be computed individually. As a consequence, this method is strongly parallelizable. However, it entails the creation of a new abstract Miter and the subsequent computation of the CSs for every configuration. Since the number of possible configurations grows exponentially as the complexity of the system increases, the main drawback of this method is the high computational complexity.

Symbolic Representation

In our running example, different allocations of patterns to basic components produce twelve different redundant architecture candidates. With more complex examples, this number can quickly grow exponentially. In order to create a more efficient method, we can leverage the similarities among alternative architectures. For example, the first two candidate architectures in Figure 6.16 differ for the third redundant component only, which is reflected in equation 6.4. This means that the Miter for the two architectures differs only for the last stage. Thus, creating a brand new Miter for each architecture - as it happens with the explicit representation - can be avoided by finding out a

single Boolean formula composed by configuration variables cfg_i that determine selected patterns, and fault variables F_i that depend on configuration and indicate whether a certain component C_i is faulty or not. Each solution of this formula encodes a CS that can be represented, via a propositional formula, as a conjunction of component faults, and the set of configurations can be represented as a disjunction of CSs. Once this formula is extracted, we can apply a recursive algorithm that extracts a symbolic reliability function that maps each configuration to its fault probability.

Minimal Cut-Sets Computation. Once we have completed the Miter composition, it allows us to generate the set of conditions that may cause the two systems to provide different outputs in presence of the same inputs, i.e., the CSs. Please remember that CS analysis is defined for coherent systems, i.e. without negation operators. Intuitively, coherence means that whenever the system has failed, no occurrence of any further basic event will ever result in a state where the system resumes functioning. We have represented the Miter as an SMT formula over input ports \vec{I} , output ports \vec{O} , fault variables \vec{F} , and TLE . Thus, the formula 5.7 describing the CSs represents the set of assignments to the fault variables such that there exists an assignment to the inputs that allows the two architectures to provide different output values. Since the Miter formula consists of input variables, output variables, and fault variables, quantifying out the inputs and outputs of each sub-component, we obtain a Boolean formula with only fault variables. The problem of extracting the CSs can be therefore encoded as an AllSMT for the theory of EUF, i.e., computing all minimal solutions with respect to the set of decision variables, as illustrated by Bozzano et al. [154]. The resulting formula is a Boolean formula consisting of fault variables, and Boolean input and output ports. The employment of the abstract Miter allows us to partition a global AllSMT(EUF) computation into a number of smaller and less complex AllSMT(EUF) applied to each CSA. Listing 6.1 shows the Python for AllSMT implementation.

Afterwards, to obtain a formula containing fault variables and configuration variables only, an additional quantifier elimination of Boolean inputs and outputs has to be performed on the global formula: to this aim we can use BDD-based projection techniques.

```

1 def allsmt(formula, to_keep_atoms: list):
2     msat = Solver(name="msat")
3     converter = msat.converter
4
5     # add the CSA formula to the solver
6     msat.add_assertion(formula)
7     result = []
8     model_counter = [0]
9     models_sec = [0]
10
11     # Invoke MathSAT APIs
12     print("[AllSMT] Compute allSMT on the formula...")
13     mathsat.msat_all_sat(msat.msat_env(),
14                          [converter.convert(atom) for atom in to_keep_atoms],
15                          # Convert the pySMT term into a MathSAT term
16                          lambda model: callback(model, converter, result, model_counter,
17                                                  start_time))
17     res_formula = Or(result)
18     print("[AllSMT] -> Done! " + str(model_counter[0]) + " models found", flush=True)
19     return res_formula

```

Listing 6.1: AllSMT computation

Reliability Function Extraction. Finally, once we have obtained the Boolean formula representing the CSs of the possible redundant architectures, we have to extract a symbolic function that maps each configuration to its failure probability. To do that we associate a probability of failure f_i to each fault variable F_i , and convert the formula of the CSs of all valid redundant configurations into a BDD-based representation. Figure 6.21 illustrates the fault variables that contribute to the reliability of the redundant alternative architecture of Figure 6.20. We can find out a Boolean formula composed by configuration variables cfg_i that determine selected patterns, and fault variables F_i that depend on configuration and indicate whether a certain component C_i is faulty or not. Since on the basis of the refinement presented in Section 5.6.2 about the sharing of fault variables of patterns allocable to the same component we can reduce the number of fault variables, the assignment of each value of probability to the corresponding probability variable depends on the configuration. The association of failure probabilities is therefore encoded as a conjunction of implications as shown in the following example.

$$\text{cfg}_1 = [0, 0] \rightarrow \left(\begin{array}{l} f_1 = 0.025 \wedge \\ f_2 = 0.030 \wedge \\ f_3 = 0.020 \end{array} \right) \wedge \text{cfg}_2 = [1] \rightarrow \left(\begin{array}{l} f_1 = 0.025 \wedge \\ f_2 = 0.023 \wedge \\ f_3 = 0.022 \wedge \\ f_4 = 0.020 \end{array} \right) \wedge \text{cfg}_3 = [0] \rightarrow \left(\begin{array}{l} f_1 = 0.022 \wedge \\ f_2 = 0.021 \wedge \\ f_3 = 0.015 \end{array} \right)$$

Similarly to the technique presented in Chapter 5, we extract the reliability function using the BDD equivalent to the Boolean formula representing the

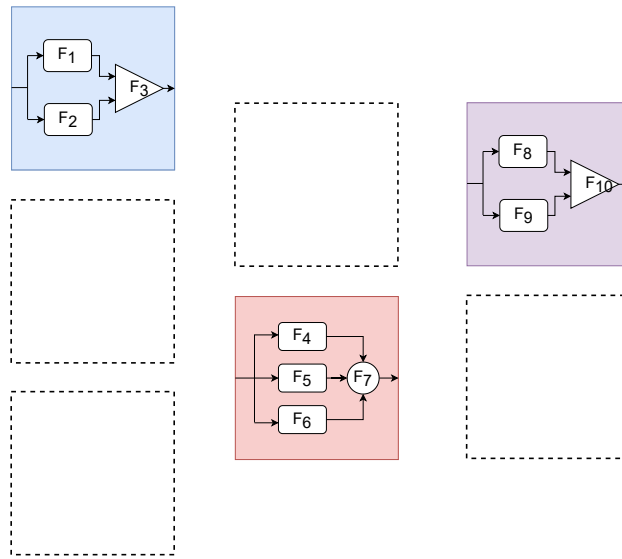


Figure 6.21: Fault variables for redundant architecture alternative 1

CSs of each configuration. Each inner node of the BDD represents a configuration or a fault variable that belongs to a certain block. Every assignment of configuration and fault variables determines a path from the root to a True-leaf, which leads to TLE. Each path from the root to a False-leaf, instead, represents an invalid assignment of configuration variables (i.e., an assignment that does not satisfy the configuration constraints) or an assignment that does not cause the TLE.

The OBDD of the DNF formula represents the MCS of the redundant system as function of the configuration and fault variables. Hence, analogously to the formula, the BDD representing its CSs contains two types of nodes: *configuration nodes* and *fault nodes*. Figure 6.22 illustrates an excerpt of the OBDD produced for the running example.

From the BDD we can calculate the fault probability of the entire system by recursively applying the formula 5.9. In case of a configuration node, evaluation of formula 5.9 translates into a *ITE* (see Figure 6.22) that basically selects the path to be followed and the fault variables involved in the computation of reliability function. The probability is then modeled by a variable f_n whose value is either equal to the probability computed in the high node or equal to the probability computed in the low node, depending on the truth value of the configuration variable. In case of a fault node, the function is assembled by simply applying the formula 5.9.

More in details, the OBDD can be built as follows.

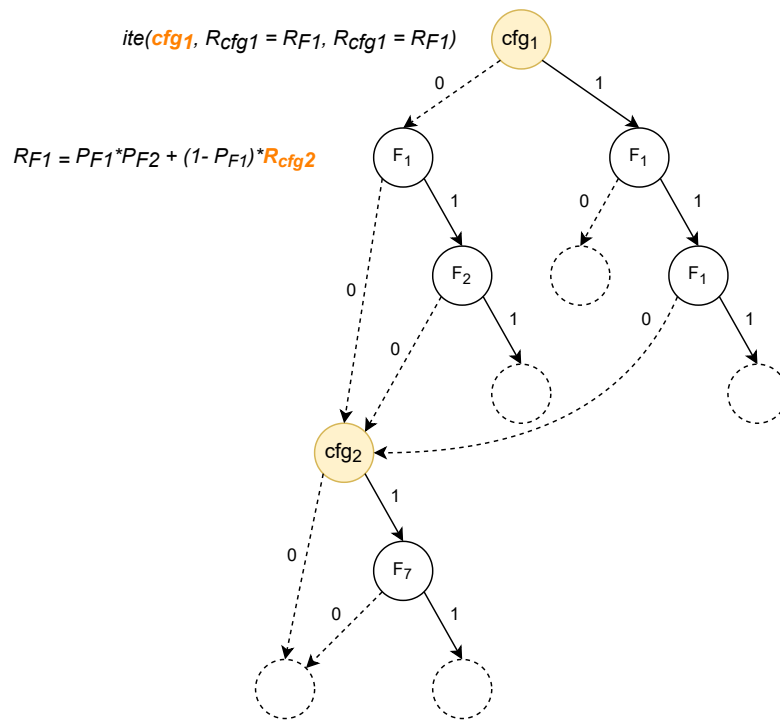


Figure 6.22: Excerpt of an OBDD encoding the CSs of the running example

- For each configuration node, tag with 1 the *if* edge and 0 the *else* edge if the variable encoded by the node has a True value, do the opposite if it has a False value.
- Traverse the OBDD from its root. Whenever a configuration node is encountered, bypass such node by adding an edge directed to the node pointed by the edge tagged with 1 and delete that configuration node. Tag each traversed fault node.
- Prune each fault node which is not tagged.

Those steps translates into Listing 6.2 that implements the traversing of the OBDD through a DFS. Figure 6.23 depicts a scenario with two configuration nodes. Bold lines represent the paths to be followed depending on the values assumed by the configuration variables. Once the failure probability of the redundant architecture A_R has been extracted, the reliability can be trivially obtained by complementing it:

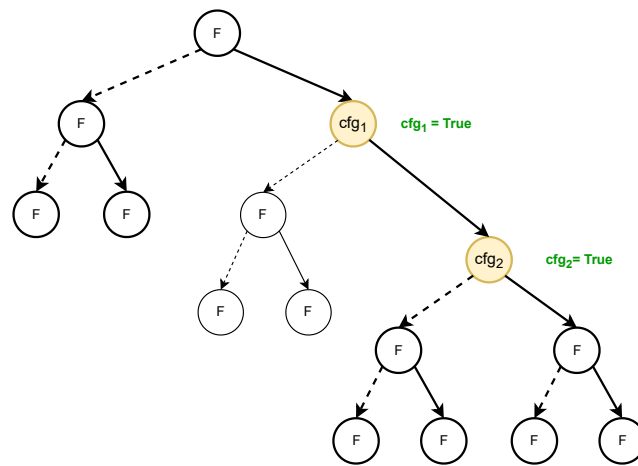
$$Rel_{AR} = 1 - f_{AR}.$$

```

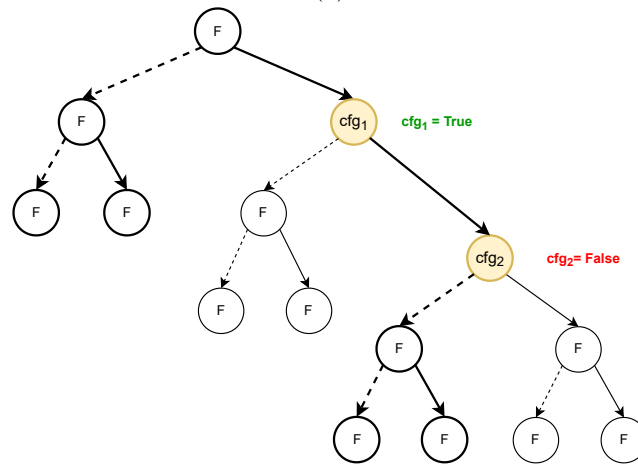
1 def extract_reliability(self):
2     """
3     Perform a DFS over a Networkx structure representing the BDD
4     :return: reliability formula
5     """
6
7     [...]
8
9     rel_formulas = []
10    # 1th dfs: Define rel variable for each fault node and ite variable for each cfg node
11    root = tree.nodes[0]
12    root_data = root['data']
13    rel = root_data.rel
14    if root_data.type != NodeType.LEAF:
15        f_data = None
16        t_data = None
17        for neighbour, data in tree.adj[0].items():
18            if data['branch'] == 0:
19                f_data = tree.nodes[neighbour]['data']
20            else:
21                t_data = tree.nodes[neighbour]['data']
22        assert f_data is not None and t_data is not None, "A BDD node not 2 neighbours"
23    if root_data.type == NodeType.FAULT:
24        rel_formulas.append(
25            self._get_rel_formula(root_data.rel, self._f_symbols2prob[root_data.var],
26                                 f_data.rel, t_data.rel))
27
28    else:
29        rel_formulas.append(
30            Ite(
31                root_data.var,
32                Equals(root_data.rel, t_data.rel),
33                Equals(root_data.rel, f_data.rel)
34            )
35        )
36
37    else:
38        assert True, "Valid or unsatisfiable formulas are not accepted"
39
40    for (s, d) in nx.dfs_edges(tree, source=0):
41        node_data = tree.nodes[d]['data']
42        branch = tree.edges[(s, d)]['branch']
43        f_data = None
44        t_data = None
45
46        if node_data.type != NodeType.LEAF:
47            for neighbour, data in tree.adj[d].items():
48                if data['branch'] == 0:
49                    f_data = tree.nodes[neighbour]['data']
50                else:
51                    t_data = tree.nodes[neighbour]['data']
52            assert f_data is not None and t_data is not None, "A node of the bdd has not 2
53            neighbours"
54
55            if node_data.type == NodeType.FAULT: # case: fault node
56                rel_formulas.append(
57                    self._get_rel_formula(node_data.rel, self._f_symbols2prob[node_data.
58                    var], f_data.rel, t_data.rel))
59
60            else: # case: cfg node
61                rel_formulas.append(
62                    Ite(
63                        node_data.var,
64                        Equals(node_data.rel, t_data.rel),
65                        Equals(node_data.rel, f_data.rel)
66                    )
67                )
68
69            else: # case: leaf
70                if node_data.value:
71                    rel_formulas.append(Equals(node_data.rel, Real(1)))
72                else:
73                    rel_formulas.append(Equals(node_data.rel, Real(0)))
74
75    print("[Extractor] Done!")
76    return And(And(rel_formulas), Equals(self._rel_symbol, rel))

```

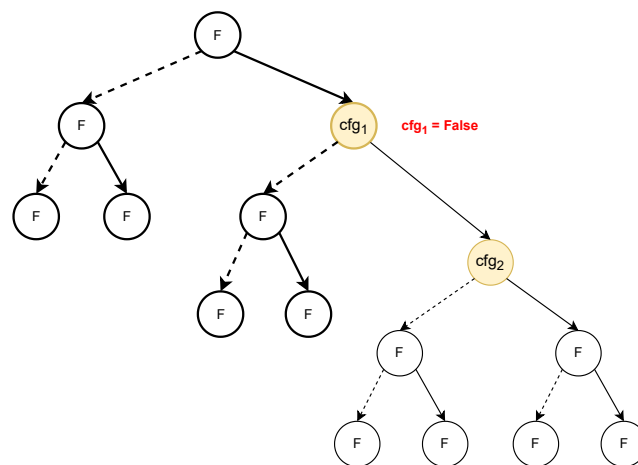
Listing 6.2: Extraction of reliability formula (excerpt)



(a)



(b)



(c)

Figure 6.23: Examples of BDD traversing: $cfg_1 = \top$ and $cfg_2 = \top$ (a), $cfg_1 = \top$ and $cfg_2 = \perp$ (b), $cfg_1 = \perp$ and $cfg_2 = \perp$ (c).

Semi-symbolic Representation

An intermediate solution could be considered, by making simplifications or "semi-symbolic" choices. For example, one could split the redundant components in two sets, and treating some of them symbolically, while the others would be made explicit. For instance, if we consider a scenario with 2^4 possible combinations, we can choose to face a single problem with four variables, or sixteen different problems each of which corresponds to a possible redundant configuration, or four problems in which each has only 2 symbolic choices possible at run-time and can choose each choice, or 4 problems each of which has only two possible symbolic choices at run-time.

To sum up, either the system is parameterized and the parameters are made explicit before the search (it corresponds to doing no search), or one of the parameters is fixed and the remaining ones are enumerated, or a single symbolic representation of the problem is used. Once we have the symbolic formula, we have the possibility to make all the available choices, ranging from an enumerative to a fully symbolic approach.

6.3.8 Assessment of Other Non-functional Parameters

Together with reliability, we also consider other non-functional requirements such as cost, power dissipation, and size, facing therefore a MOOP. The assessment of new design objectives may vary from case to case, but since the main focus of our work is reliability, for simplicity we suppose that the other requirements are cumulative. In this case, the assessment is quite simple as we can consider the cost, power dissipation, and size of the redundant system equal to the sum of the cost, power dissipation, and size of the single composing redundant patterns. This is not a restriction for our methodology. It could easily accommodate more complex functions. Once the objective function of each non-functional parameter is computed, the optimizer has the task to explore the design space in order to finding the Pareto optimal solutions. Likewise, our methodology allow us to address the problem of synthesizing system architectures to minimize one or more cost functions while guaranteeing the desired reliability, i.e., reliability is a constraint and optimization is performed on other metrics.

6.3.9 Optimization

To explore the design space and find the allocations that optimize the objective functions of the redundant system, we use the Z3 solver, taking advantage of its highly optimized OMT solver, without having to dive into its implementation. Considering that the number of configurations is combinatorial with respect to the size of the architecture, the resulting function defining the space of the design alternatives is very complex. Since vanilla OMT is not very efficient with it, making the problems intractable when the number of components or patterns is of the order of dozens, we adopt a trade-off between fully symbolic and enumerative methods, which consists in extracting the symbolic reliability function of each parameter, and then using it to retrieve the actual values of each configuration, in order to build a function that explicitly defines all the design points of the space. This approach makes those problems affordable. More in general, we can parameterize the system and choose between a single symbolic representation of the problem (leaving the choice at execution time), a representation where a parameter is fixed and the remaining ones are enumerated, and a representation in which the parameters are made explicit before the search (it corresponds to doing no search), going from a fully symbolic, to a semi-symbolic, to an enumerative approach, which is strongly parallelizable.

6.3.10 Improvements and Refinements

In the following, we present two strategies in order to enhance our method. The first strategy leverages similarities among the set of redundant architectures we deal with, improving the performance of our method by reducing the number of CSA needed to compose the Miter. The second strategy aims at reducing the size of the OBDD representing the formula of the CSs, in order to decrease the time needed for quantifier elimination.

Minimal Cut-Sets Computation of Symbolic representations

Many candidate redundant architectures differ for only one component, and more in general they share some similarities. This propriety can be exploited in order to make the assessment of reliability more efficient. Get back to our running example of Figure 6.12, with a slightly modification, as reported in Table 6.4. Basically, components C_1 and C_2 have two suitable redundant

patterns that are different instances of the same type, namely a TMR_V111 , as illustrated in Figure 6.24.

Table 6.4: Library of patterns for example system in Figure 6.12

Component	Pattern	Type
C_1	P_1	TMR with 1 Voter (TMR_V111)
	P_2	TMR with 1 Voter (TMR_V111)
	P_3	TMR with 3 Voters (TMR_V123)
C_2	P_4	TMR with 1 Voter (TMR_V111)
	P_5	TMR with 1 Voter (TMR_V111)
C_3	P_6	TMR with 1 Voter (TMR_V111)
	P_7	TMR with 3 Voters (TMR_V123)

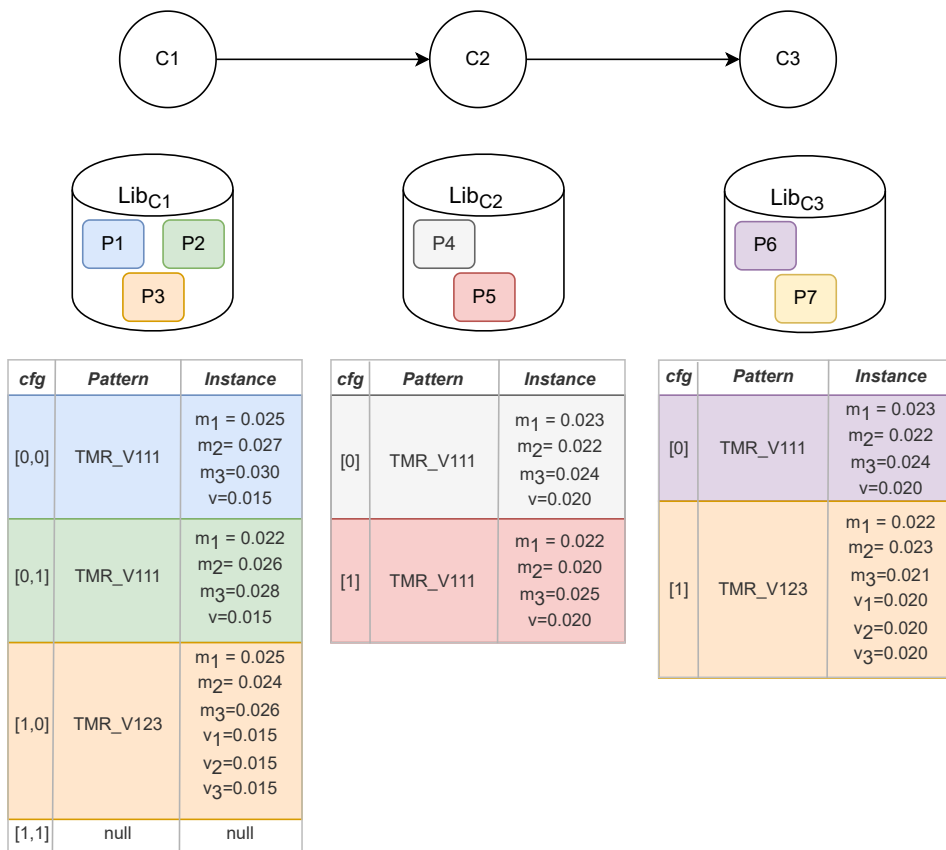


Figure 6.24: Example of basic (non-redundant) system and a library including instances of the same pattern type.

The Miter composition is illustrated in Figure 6.25. It allows us to encode the set of alternative redundant architectures in a single formula. Please note that since in this step the aim is finding the CSs, we are only interested in

the internal behavior of each single pattern, overlooking for the moment their probabilities of failure. Thus, we only consider the pattern types for each component, referencing them through a disjunction of configuration variables that identify the same pattern type. For your better understanding, in the example of Figure 6.24, the pattern type TMR_V111 valid for the component C_1 is referenced by the following formula:

$$cfg_1 = [0, 0] \vee cfg_1 = [0, 1]$$

and the pattern type TMR_V111 valid for the component C_2 is referenced by the following formula:

$$cfg_2 = [0] \vee cfg_2 = [1]$$

The Miter is fed with the Boolean constant True because we assume that the input of the overall architecture is nominal. Blue arrows represent the selection of redundant pattern for the upstream component. Downstream CSA_2 there is no selection as the library of redundant patterns available for component C_2 include only the pattern type TMR_V111 . Please note that the CSA stage in figure have only one input and output ports. Obviously, redundant pattern may have multiple input and output ports, and the multiplexer has to select all outputs of a given redundant pattern. Compared to the abstract Miter of Figure 6.17, this new Miter has multiple CSA modules for a single component. The specification of which CSA is active at time (and consequently which is the output of each stage) is performed by configuration variables that select the redundant pattern assigned to each component.

As we have done before, selected outputs are linked to the inputs of every subsequent redundant component via linking constraints. In our running example this translates into the following constraints:

$$(cfg_1 = [0, 0] \vee cfg_1 = [0, 1]) \rightarrow (CSA_{11}.out \leftrightarrow CSA_2.in) \cap \quad (6.5)$$

$$(cfg_1 = [1, 0]) \rightarrow (CSA_{12}.out \leftrightarrow CSA_2.in) \quad (6.6)$$

and

$$(cfg_2 = [0] \vee cfg_2 = [1]) \rightarrow \left(\begin{array}{l} CSA_2.out \leftrightarrow CSA_{31}.in \cap \\ CSA_2.out \leftrightarrow CSA_{32}.in \end{array} \right)$$

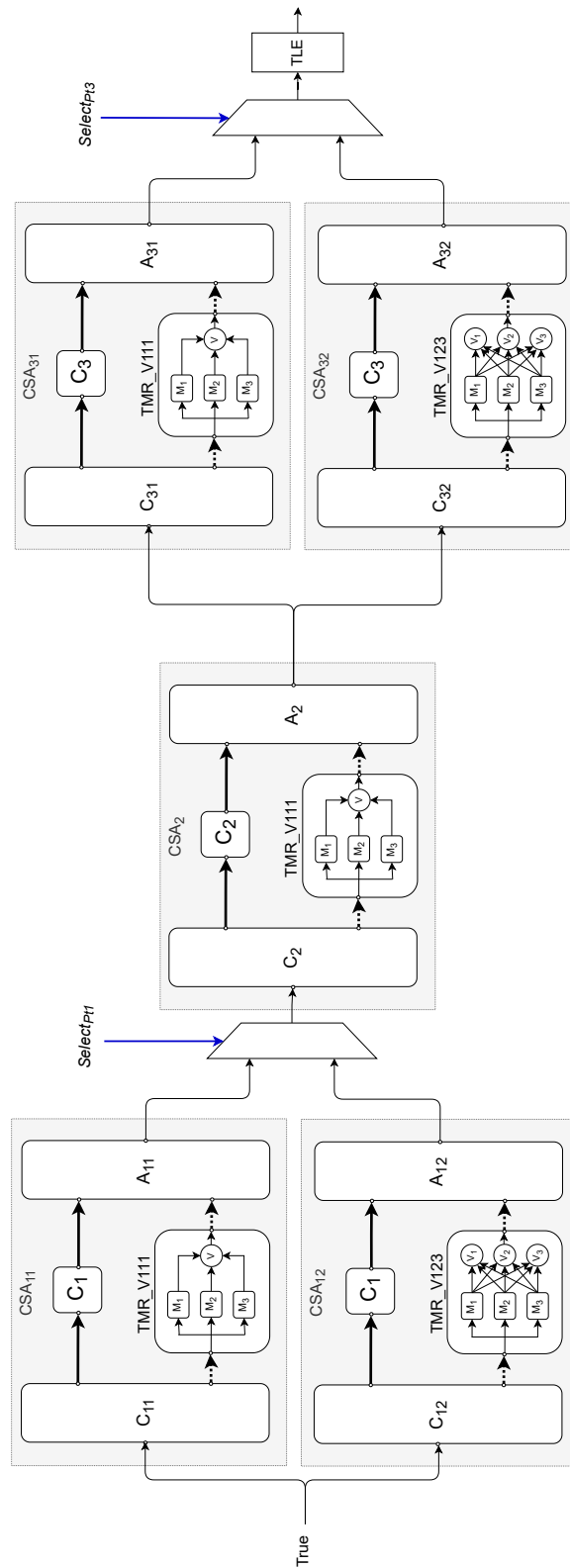


Figure 6.25: Combinatorial abstract Miter composition for architecture of Figure 6.24

Also in this composition, incompatible patterns that cannot be connected are excluded by adding compatibility constraints. As in the previous Miter, each model of the formula representing the combinatorial abstract Miter models a possible state of the system that triggers the TLE. The main property of this composition is that it is possible to encode all deviations from the nominal behavior of each possible redundant system architecture using the configuration variables.

Choosing Optimal Variable ordering

The main disadvantage of BDDs is low scalability: the size of a BDD corresponding to a propositional formula can be exponential in the number of variables. To employ BDDs in SMT solving we apply quantifiers to the formula. Considering existential quantification reduces size of a BDD as it decreases the number of its variables. Usually, finding good variable orderings is an important task in CSPs and SAT, but it becomes fundamental when applying symbolic search. As additional refinement to our method, we can build the BDD using a specific variable ordering, as the ordering has influence on the BDD size. A good ordering gives a concise BDD form. A bad ordering may lead to an explosion in the size of the BDD used to represent the fault tree. A fault tree with x_1, x_2, \dots, x_n basic events has its equivalent boolean function $f(x_1, x_2, \dots, x_n)$ and vice versa. However, the order that yields the smallest BDD representation does not necessarily have the best performance (due to BDD cache effects). The run time depends greatly on the characteristics of the inputs. Finding the optimal order for a given function is an NP-complete problem, and it is out of our scope. However, we are going to consider three different variable orderings, in order to show the impacts on the computation times.

For sake of simplicity, consider the example system in Figure 6.26 composed of only two components, C_1 and C_2 , each one with two valid patterns for redundancy, P_1 and P_2 for the former P_3 and P_4 for the latter. Suppose that P_1 and P_3 are made of two sub-components and the entire components fail if both sub-components fail. P_2 and P_4 are modeled with a single fault variable that determine the failure.

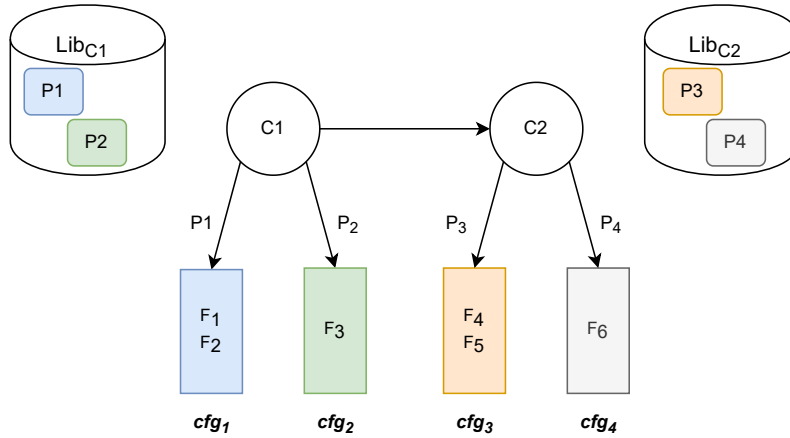


Figure 6.26: Example system of two components, each with one two redundant pattern.

The system will fail if C_1 or C_2 will fail. The first occurrence can be modeled with the following formula:

$$(cfg_1 \wedge (F_1 \wedge F_2)) \vee (cfg_2 \wedge F_3) \quad (6.7)$$

Similarly, the second occurrence can be modeled with the following formula:

$$(cfg_3 \wedge (F_4 \wedge F_5)) \vee (cfg_4 \wedge F_6) \quad (6.8)$$

Putting all together, the formula describing the CSs is the following:

$$(cfg_1 \wedge (F_1 \wedge F_2)) \vee (cfg_2 \wedge F_3) \vee (cfg_3 \wedge (F_4 \wedge F_5)) \vee (cfg_4 \wedge F_6) \leftrightarrow TLE \quad (6.9)$$

Equation 6.9 states that the system depicted in Figure 6.26 will fail if one of the two composing components will fail, and that the faulty variables involved depend on the redundant pattern allocated. In case of component C_1 , faulty variables will be F_1 and F_2 if the pattern selected for redundancy is P_1 , it will be F_3 if the pattern selected is P_2 . For component C_2 , faulty variables will be F_4 and F_5 if the pattern selected for redundancy is P_3 , it will be F_6 if the pattern selected is P_4 . To facilitate the understanding, please note that in the above example we did not employ the refinements introduced in Section 5.6.2. Furthermore, we used a 1-hot encoding. In the following, we examine how the BDD representing equation 6.9 looks like for different variable orderings.

Ordering 1: All Configuration Variables Precede Fault Variables.

The OBDD is organized in a first part that is a kind of configuration DAG, followed by a second part that is composed of fault variables only. Hence, the configuration variables are on top of the BDD, and fault variables are in the bottom part. Figure 6.27 shows the resulting BDD of the example in Figure 6.26. Orange nodes are configuration nodes, while white nodes are fault nodes.

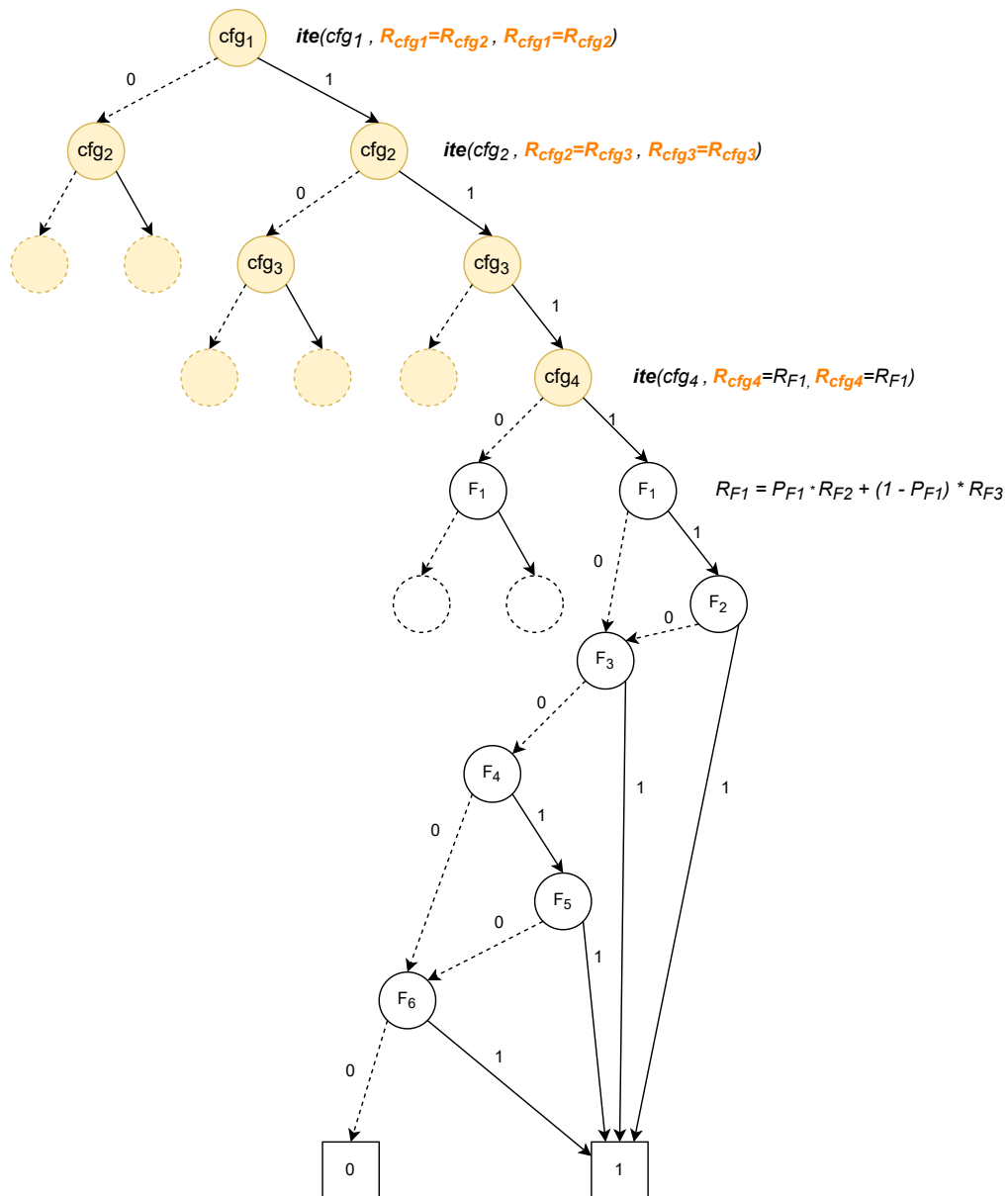


Figure 6.27: Partial BDD for example in Figure 6.26, using ordering with all configuration variables on top.

Ordering 2: Arbitrary, no Assumption. Assign arbitrary total ordering to variables. Figure 6.28 and Figure 6.29 shows the resulting BDD of two different arbitrary orderings. We can try different heuristics to find an order that keeps the OBDD size manageable. For example, we can enable dynamic variable ordering using the SIFT heuristic [297] or other existing algorithms.

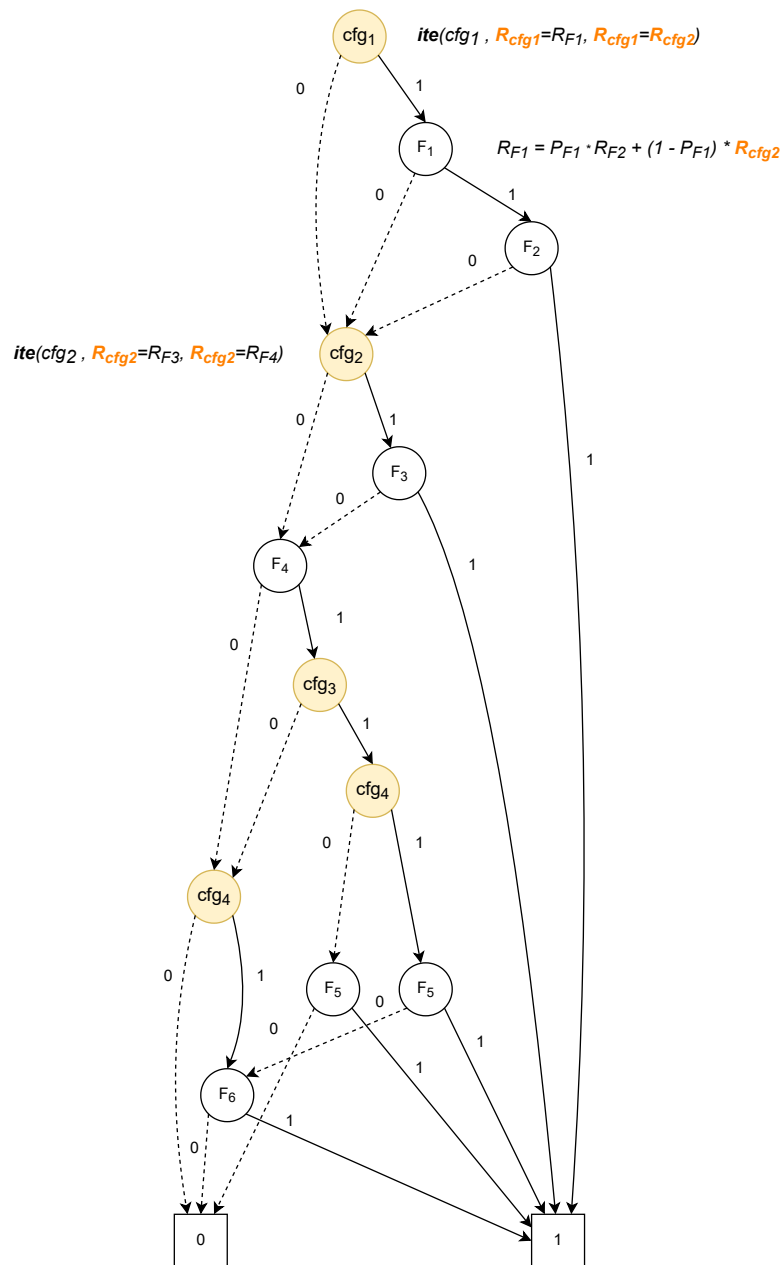


Figure 6.28: BDD for example in Figure 6.26, using arbitrary ordering of variables.

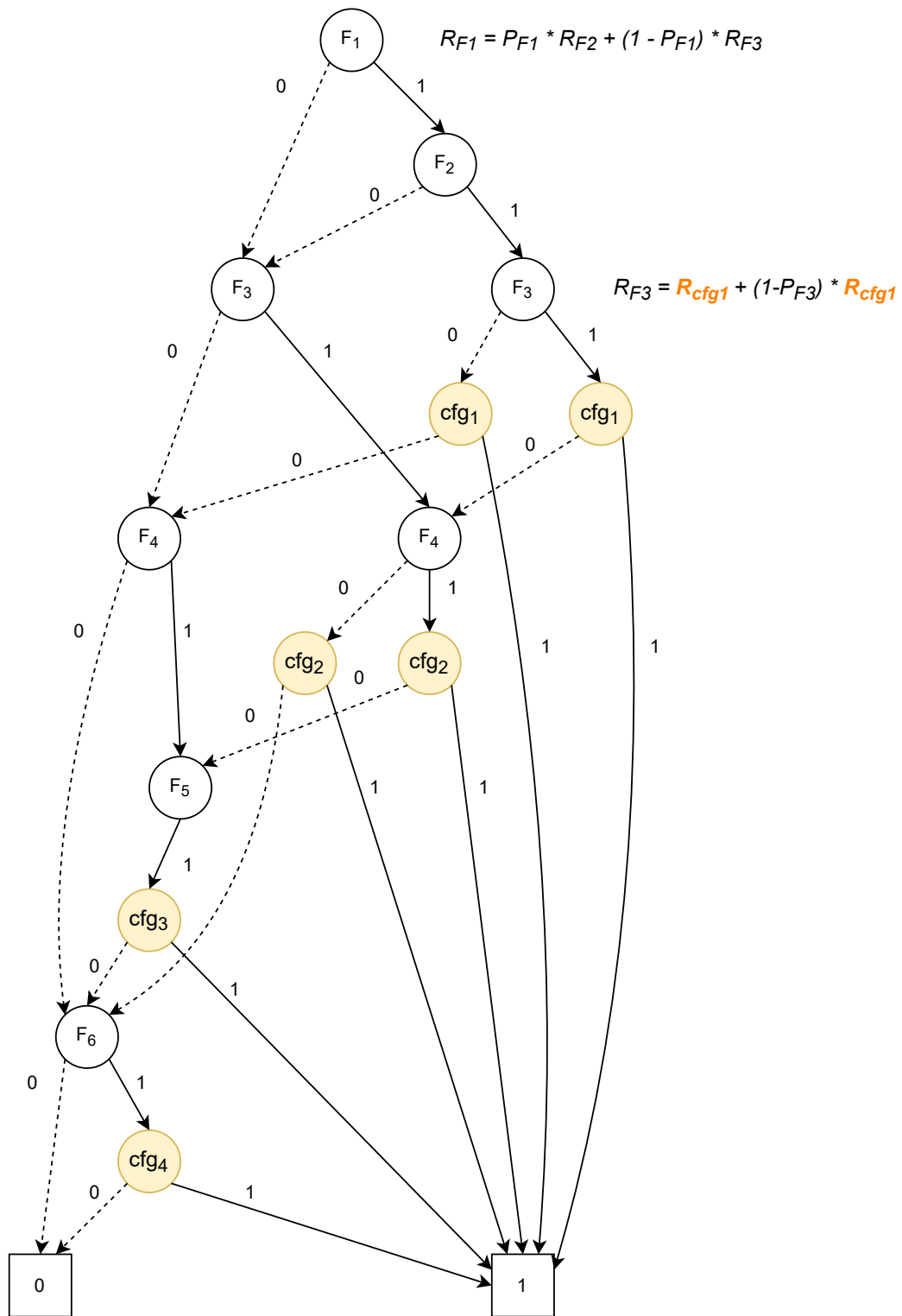


Figure 6.29: BDD for example in Figure 6.26, using an alternative arbitrary ordering of variables.

Ordering 3: Driven by Architecture. The ordering is static and is based on the topology of the basic architecture. The first level of variables is composed by configuration variables of the first component C_1 , followed by the faults and output predicates of the two patterns applicable to the component. Similarly, the second level includes the configuration variables followed by faults and output predicates of component C_2 . And so on for all components of given architecture. The resulting BDD construction for example of Figure 6.26 is illustrated in Figure 6.30.

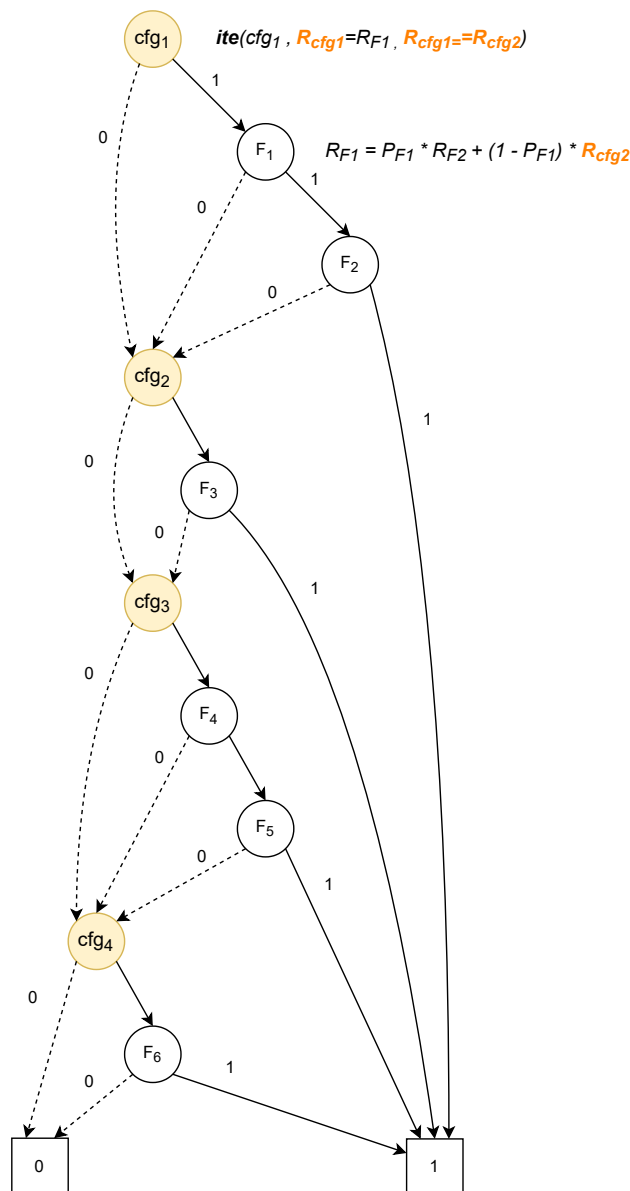


Figure 6.30: BDD for example in Figure 6.26, using an ordering that follows architecture's topology.

cfg4: 1, F1: 0, F3: 0, F4: 0, F6: 1}, {cfg1: 1, cfg2: 1, cfg3: 1, cfg4: 1, F1: 0, F3: 0, F4: 1, F5: 0, F6: 1},
 {cfg1: 1, cfg2: 1, cfg3: 1, cfg4: 1, F1: 0, F3: 0, F4: 1, F5: 1}, {cfg1: 1, cfg2: 1, cfg3: 1, cfg4: 1, F1: 0, F3:
 1}, {cfg1: 1, cfg2: 1, cfg3: 1, cfg4: 1, F1: 1, F2: 0, F3: 0, F4: 0, F6: 1}, {cfg1: 1, cfg2: 1, cfg3: 1, cfg4: 1,
 F1: 1, F2: 0, F3: 0, F4: 1, F5: 0, F6: 1}, {cfg1: 1, cfg2: 1, cfg3: 1, cfg4: 1, F1: 1, F2: 0, F3: 0, F4: 1,
 F5: 1}, {cfg1: 1, cfg2: 1, cfg3: 1, cfg4: 1, F1: 1, F2: 0, F3: 1}, {cfg1: 1, cfg2: 1, cfg3: 1, cfg4: 1, F1: 1, F2: 1}

Satisfy assignments using ordering 2 follow:

{cfg1: 0, cfg2: 0, F4: 0, cfg4: 1, F6: 1}, {cfg1: 0, cfg2: 0, F4: 1, cfg3: 0, cfg4: 1, F6: 1}, {cfg1: 0, cfg2:
 0, F4: 1, cfg3: 1, cfg4: 0, F5: 1}, {cfg1: 0, cfg2: 0, F4: 1, cfg3: 1, cfg4: 1, F5: 0, F6: 1}, {cfg1: 0, cfg2:
 0, F4: 1, cfg3: 1, cfg4: 1, F5: 1}, {cfg1: 0, cfg2: 1, F3: 0, F4: 0, cfg4: 1, F6: 1}, {cfg1: 0, cfg2: 1, F3:
 0, F4: 1, cfg3: 0, cfg4: 1, F6: 1}, {cfg1: 0, cfg2: 1, F3: 0, F4: 1, cfg3: 1, cfg4: 0, F5: 1}, {cfg1: 0, cfg2:
 1, F3: 0, F4: 1, cfg3: 1, cfg4: 1, F5: 0, F6: 1}, {cfg1: 0, cfg2: 1, F3: 0, F4: 1, cfg3: 1, cfg4: 1, F5: 1},
 {cfg1: 0, cfg2: 1, F3: 1}, {cfg1: 1, F1: 0, cfg2: 0, F4: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 0, cfg2: 0, F4: 1,
 cfg3: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 0, cfg2: 0, F4: 1, cfg3: 1, cfg4: 0, F5: 1}, {cfg1: 1, F1: 0, cfg2: 0,
 F4: 1, cfg3: 1, cfg4: 1, F5: 0, F6: 1}, {cfg1: 1, F1: 0, cfg2: 0, F4: 1, cfg3: 1, cfg4: 1, F5: 1}, {cfg1: 1,
 F1: 0, cfg2: 1, F3: 0, F4: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 0, cfg2: 1, F3: 0, F4: 1, cfg3: 0, cfg4: 1, F6:
 1}, {cfg1: 1, F1: 0, cfg2: 1, F3: 0, F4: 1, cfg3: 1, cfg4: 0, F5: 1}, {cfg1: 1, F1: 0, cfg2: 1, F3: 0, F4: 1,
 cfg3: 1, cfg4: 1, F5: 0, F6: 1}, {cfg1: 1, F1: 0, cfg2: 1, F3: 0, F4: 1, cfg3: 1, cfg4: 1, F5: 1}, {cfg1: 1,
 F1: 0, cfg2: 1, F3: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 0, F4: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2:
 0, F4: 1, cfg3: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 0, F4: 1, cfg3: 1, cfg4: 0, F5: 1}, {cfg1: 1,
 F1: 1, F2: 0, cfg2: 0, F4: 1, cfg3: 1, cfg4: 1, F5: 0, F6: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 0, F4: 1, cfg3:
 1, cfg4: 1, F5: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 1, F3: 0, F4: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 1, F2: 0,
 cfg2: 1, F3: 0, F4: 1, cfg3: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 1, F3: 0, F4: 1, cfg3: 1, cfg4: 0,
 F5: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 1, F3: 0, F4: 1, cfg3: 1, cfg4: 1, F5: 0, F6: 1}, {cfg1: 1, F1: 1, F2:
 0, cfg2: 1, F3: 0, F4: 1, cfg3: 1, cfg4: 1, F5: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 1, F3: 1}, {cfg1: 1, F1: 1, F2: 1}

Satisfy assignments using a different ordering 2 follow:

{F1: 0, F3: 0, F4: 0, F6: 1, cfg4: 1}, {F1: 0, F3: 0, F4: 1, F5: 0, F6: 1, cfg4: 1}, {F1: 0, F3: 0, F4: 1, F5:
 1, cfg3: 0, F6: 1, cfg4: 1}, {F1: 0, F3: 0, F4: 1, F5: 1, cfg3: 1}, {F1: 0, F3: 1, F4: 0, cfg2: 0, F6: 1, cfg4:
 1}, {F1: 0, F3: 1, F4: 0, cfg2: 1}, {F1: 0, F3: 1, F4: 1, cfg2: 0, F5: 0, F6: 1, cfg4: 1}, {F1: 0, F3: 1, F4:
 1, cfg2: 0, F5: 1, cfg3: 0, F6: 1, cfg4: 1}, {F1: 0, F3: 1, F4: 1, cfg2: 0, F5: 1, cfg3: 1}, {F1: 0, F3: 1, F4:
 1, cfg2: 1}, {F1: 1, F2: 0, F3: 0, F4: 0, F6: 1, cfg4: 1}, {F1: 1, F2: 0, F3: 0, F4: 1, F5: 0, F6: 1, cfg4: 1},
 {F1: 1, F2: 0, F3: 0, F4: 1, F5: 1, cfg3: 0, F6: 1, cfg4: 1}, {F1: 1, F2: 0, F3: 0, F4: 1, F5: 1, cfg3: 1},
 {F1: 1, F2: 0, F3: 1, F4: 0, cfg2: 0, F6: 1, cfg4: 1}, {F1: 1, F2: 0, F3: 1, F4: 0, cfg2: 1}, {F1: 1, F2: 0,
 F3: 1, F4: 1, cfg2: 0, F5: 0, F6: 1, cfg4: 1}, {F1: 1, F2: 0, F3: 1, F4: 1, cfg2: 0, F5: 1, cfg3: 0, F6: 1, cfg4:
 1}, {F1: 1, F2: 0, F3: 1, F4: 1, cfg2: 0, F5: 1, cfg3: 1}, {F1: 1, F2: 0, F3: 1, F4: 1, cfg2: 1}, {F1: 1, F2:
 1, F3: 0, cfg1: 0, F4: 0, F6: 1, cfg4: 1}, {F1: 1, F2: 1, F3: 0, cfg1: 0, F4: 1, F5: 0, F6: 1, cfg4: 1}, {F1:

1, F2: 1, F3: 0, cfg1: 0, F4: 1, F5: 1, cfg3: 0, F6: 1, cfg4: 1}, {F1: 1, F2: 1, F3: 0, cfg1: 0, F4: 1, F5: 1, cfg3: 1}, {F1: 1, F2: 1, F3: 0, cfg1: 1}, {F1: 1, F2: 1, F3: 1, cfg1: 0, F4: 0, cfg2: 0, F6: 1, cfg4: 1}, {F1: 1, F2: 1, F3: 1, cfg1: 0, F4: 0, cfg2: 1}, {F1: 1, F2: 1, F3: 1, cfg1: 0, F4: 1, cfg2: 0, F5: 0, F6: 1, cfg4: 1}, {F1: 1, F2: 1, F3: 1, cfg1: 0, F4: 1, cfg2: 0, F5: 1, cfg3: 0, F6: 1, cfg4: 1}, {F1: 1, F2: 1, F3: 1, cfg1: 0, F4: 1, cfg2: 0, F5: 1, cfg3: 1}, {F1: 1, F2: 1, F3: 1, cfg1: 0, F4: 1, cfg2: 1}, {F1: 1, F2: 1, F3: 1, cfg1: 1}

Satisfy assignments using ordering 3 follow:

{cfg1: 0, cfg2: 0, cfg3: 0, cfg4: 1, F6: 1}, {cfg1: 0, cfg2: 0, cfg3: 1, F4: 0, cfg4: 1, F6: 1}, {cfg1: 0, cfg2: 0, cfg3: 1, F4: 1, F5: 0, cfg4: 1, F6: 1}, {cfg1: 0, cfg2: 0, cfg3: 1, F4: 1, F5: 1}, {cfg1: 0, cfg2: 1, F3: 0, cfg3: 0, cfg4: 1, F6: 1}, {cfg1: 0, cfg2: 1, F3: 0, cfg3: 1, F4: 0, cfg4: 1, F6: 1}, {cfg1: 0, cfg2: 1, F3: 0, cfg3: 1, F4: 1, F5: 0, cfg4: 1, F6: 1}, {cfg1: 0, cfg2: 1, F3: 0, cfg3: 1, F4: 1, F5: 1}, {cfg1: 0, cfg2: 1, F3: 1}, {cfg1: 1, F1: 0, cfg2: 0, cfg3: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 0, cfg2: 0, cfg3: 1, F4: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 0, cfg2: 0, cfg3: 1, F4: 1, F5: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 0, cfg2: 0, cfg3: 1, F4: 1, F5: 1}, {cfg1: 1, F1: 0, cfg2: 1, F3: 0, cfg3: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 0, cfg2: 1, F3: 0, cfg3: 1, F4: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 0, cfg2: 1, F3: 0, cfg3: 1, F4: 1, F5: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 0, cfg2: 1, F3: 0, cfg3: 1, F4: 1, F5: 1}, {cfg1: 1, F1: 0, cfg2: 1, F3: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 0, cfg3: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 0, cfg3: 1, F4: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 0, cfg3: 1, F4: 1, F5: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 0, cfg3: 1, F4: 1, F5: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 1, F3: 0, cfg3: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 1, F3: 0, cfg3: 1, F4: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 1, F3: 0, cfg3: 1, F4: 1, F5: 0, cfg4: 1, F6: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 1, F3: 0, cfg3: 1, F4: 1, F5: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 1, F3: 1}, {cfg1: 1, F1: 1, F2: 1}

Using Binary Encoding to Encode Configuration Variables

As already stated, for the construction of the BDDs of Figure 6.32, we used a 1-hot encoding. Using a binary encoding, we could employ only two configuration variables, mapped as follows:

$$cfg_1, cfg_2 \rightarrow cfg_1 = [0, 1]$$

$$cfg_3, cfg_4 \rightarrow cfg_2 = [0, 1]$$

Rewriting the equation 6.9 as follows:

$$(cfg_1 \wedge (F_1 \wedge F_2)) \vee (\neg cfg_1 \wedge F_3) \vee (cfg_2 \wedge (F_4 \wedge F_5)) \vee (\neg cfg_2 \wedge F_6) \leftrightarrow TLE \quad (6.10)$$

The resulting BDDs are simplified, as illustrated in Figure 6.33.

The simplification is also visible from the length of satisfy assignments. Satisfy assignments using ordering 1 follow:

{cfg1: 0, cfg2: 0, F3: 0, F6: 1}, {cfg1: 0, cfg2: 0, F3: 1}, {cfg1: 0, cfg2: 1, F3: 0, F4: 1, F5: 1}, {cfg1: 0, cfg2: 1, F3: 1}, {cfg1: 1, cfg2: 0, F1: 0, F6: 1}, {cfg1: 1, cfg2: 0, F1: 1, F2: 0, F6: 1}, {cfg1: 1, cfg2: 0, F1: 1, F2: 1}, {cfg1: 1, cfg2: 1, F1: 0, F4: 1, F5: 1}, {cfg1: 1, cfg2: 1, F1: 1, F2: 0, F4: 1, F5: 1}, {cfg1: 1, cfg2: 1, F1: 1, F2: 1}

Satisfy assignments using ordering 2 follow:

{cfg1: 0, cfg2: 0, F3: 0, F6: 1}, {cfg1: 0, cfg2: 0, F3: 1}, {cfg1: 0, cfg2: 1, F3: 0, F4: 1, F5: 1}, {cfg1: 0, cfg2: 1, F3: 1}, {cfg1: 1, cfg2: 0, F1: 0, F6: 1}, {cfg1: 1, cfg2: 0, F1: 1, F2: 0, F6: 1}, {cfg1: 1, cfg2: 0, F1: 1, F2: 1}, {cfg1: 1, cfg2: 1, F1: 0, F4: 1, F5: 1}, {cfg1: 1, cfg2: 1, F1: 1, F2: 0, F4: 1, F5: 1}, {cfg1: 1, cfg2: 1, F1: 1, F2: 1}

Satisfy assignments using a different ordering 2 follow:

{F1: 0, F3: 0, F4: 0, cfg2: 0, F6: 1}, {F1: 0, F3: 0, F4: 1, cfg2: 0, F6: 1}, {F1: 0, F3: 0, F4: 1, cfg2: 1, F5: 1}, {F1: 0, F3: 1, cfg1: 0}, {F1: 0, F3: 1, cfg1: 1, F4: 0, cfg2: 0, F6: 1}, {F1: 0, F3: 1, cfg1: 1, F4: 1, cfg2: 0, F6: 1}, {F1: 0, F3: 1, cfg1: 1, F4: 1, cfg2: 1, F5: 1}, {F1: 1, F2: 0, F3: 0, F4: 0, cfg2: 0, F6: 1}, {F1: 1, F2: 0, F3: 0, F4: 1, cfg2: 0, F6: 1}, {F1: 1, F2: 0, F3: 0, F4: 1, cfg2: 1, F5: 1}, {F1: 1, F2: 0, F3: 1, cfg1: 0}, {F1: 1, F2: 0, F3: 1, cfg1: 1, F4: 0, cfg2: 0, F6: 1}, {F1: 1, F2: 0, F3: 1, cfg1: 1, F4: 1, cfg2: 0, F6: 1}, {F1: 1, F2: 0, F3: 1, cfg1: 1, F4: 1, cfg2: 1, F5: 1}, {F1: 1, F2: 1, F3: 0, cfg1: 0, F4: 0, cfg2: 0, F6: 1}, {F1: 1, F2: 1, F3: 0, cfg1: 0, F4: 1, cfg2: 0, F6: 1}, {F1: 1, F2: 1, F3: 0, cfg1: 0, F4: 1, cfg2: 1, F5: 1}, {F1: 1, F2: 1, F3: 0, cfg1: 1}, {F1: 1, F2: 1, F3: 1}

Satisfy assignments using ordering 3 follow:

{cfg1: 0, cfg2: 0, F3: 0, F6: 1}, {cfg1: 0, cfg2: 0, F3: 1}, {cfg1: 0, cfg2: 1, F3: 0, F4: 1, F5: 1}, {cfg1: 0, cfg2: 1, F3: 1}, {cfg1: 1, F1: 0, cfg2: 0, F6: 1}, {cfg1: 1, F1: 0, cfg2: 1, F4: 1, F5: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 0, F6: 1}, {cfg1: 1, F1: 1, F2: 0, cfg2: 1, F4: 1, F5: 1}, {cfg1: 1, F1: 1, F2: 1}

As a final remark, note that ordering 1 allows us to easily identify the different sub-trees representing the CSs of each configuration. With static or dynamic ordering, configuration variables are positioned in different ways along the tree, making it hard to identify the BDD representing the CSs of each configuration.

The detailed steps of the exact method method proposed are reported in Algorithm 2.

Algorithm 2 DSE general framework

- 1: **Input 1:** system architecture model
 - 2: **Input 2:** library of redundant patterns
 - 3: **Input 3:** fault model
 - 4: **Input 4:** design objectives
 - 5: **[OPT] Input 5:** additional constraints (with local or global scope)
 - 6: **Output:** set of optimum redundant architectures
 - 7:

 - 8: **Phase 1 - Modeling**
 - 9: Define *configuration variables* $cfg_i = (C_i, P_j)$ and *fault variables* F_i
 - 10: Define the *behaviors* of the components through SMT constraints
 - a: For each CSA use basic version as nominal behavior
 - b: For each C_i , find the P_j with highest number F_i for applying variable sharing
 - 11: Define the *linking constraints* ▷ SMT formula of the redundant architectures
 - 12: Define *configuration constraints* $\text{len}(cfg_i) = \lceil \log_2(\text{len}(lib_{C_i})) \rceil$
 - 13: Define the *compatibility constraints* through SMT constraints
 - 14: **Phase 2 - Assessment**
 - 15: Miter composition
 - 16: Computation of MCSs ▷ AllSMT computation using MathSAT solver
 - 17: Caching the resulting formula
 - 18: Conversion of the formula into a BDD representation.
 - 19: BDD-based quantifier elimination.
 - 20: Symbolic Reliability function extraction ▷ by BDD traversing
 - 21: Assessment of other non-functional parameters
 - 22: **Phase 3 - Optimization**
 - 23: **if** approach = ENUMERATIVE **then**
 - 24: Compose a new Miter for each alternative
 - 25: Compute the MCS for each alternative
 - 26: Create a mapping *configuration* \rightarrow *cost functions*
 - 27: **else if** approach = SYMBOLIC **then**
 - 28: Parameterize the system
 - 29: Compute the values at run-time ▷ using Z3 solver
 - 30: **else** approach = HYBRID
 - 31: Perform semi-symbolic choices
 - 32: **end if**
-

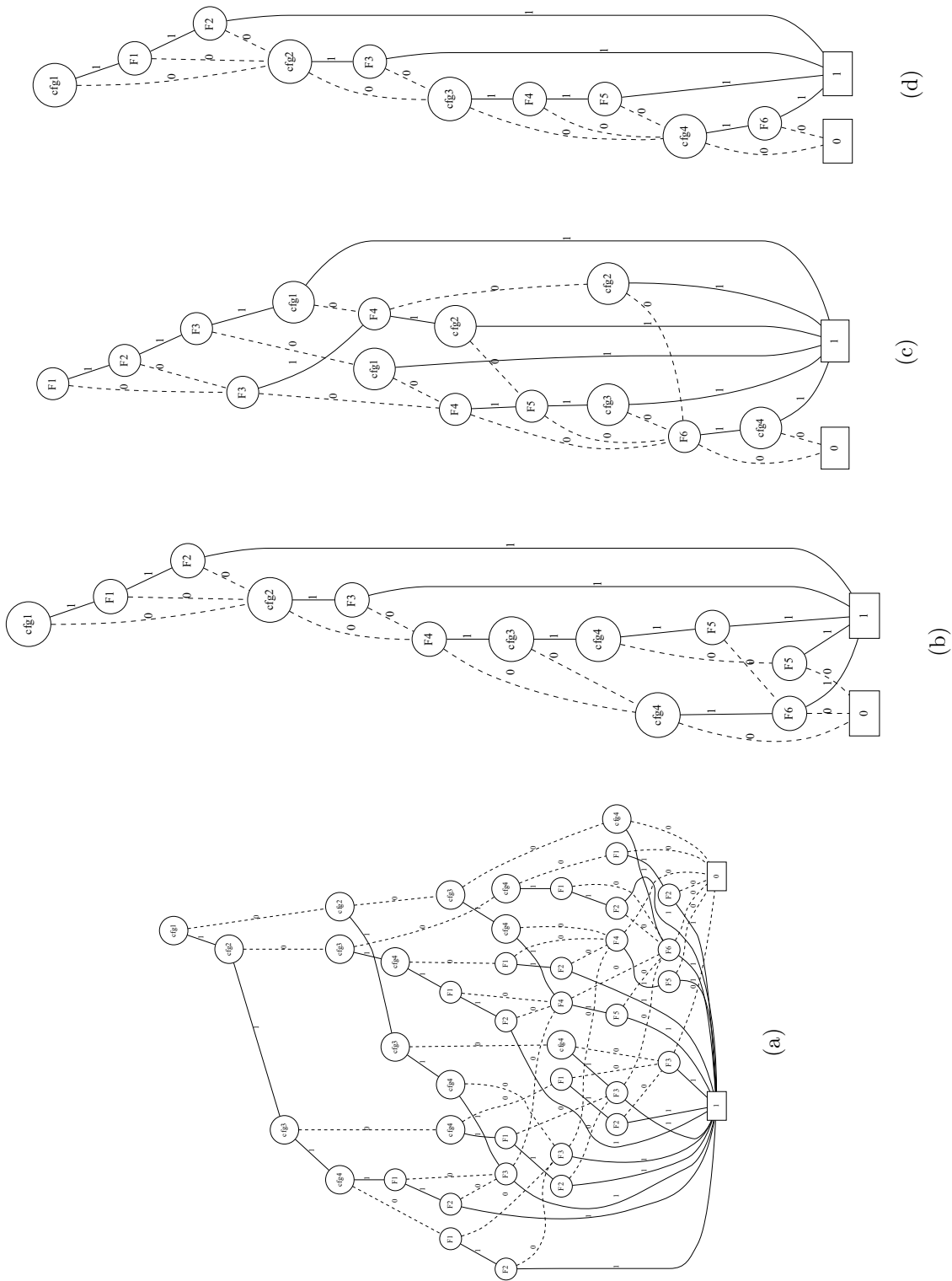


Figure 6.32: BDD of example in Figure 6.26 resulting from different orderings: all variables on top (a), arbitrary assignment (b), alternative arbitrary assignment (c), driven by architecture's topology (d).

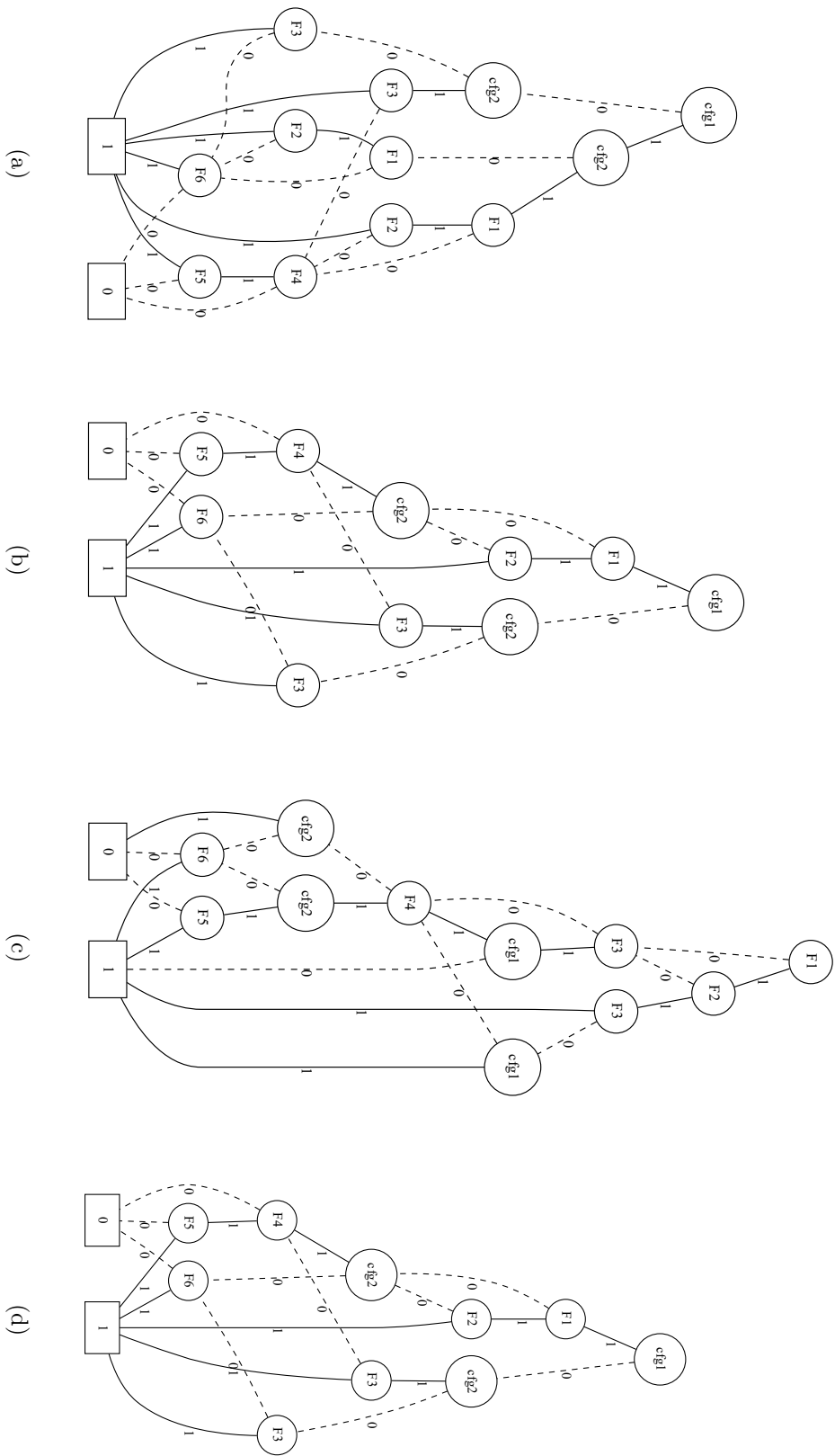


Figure 6.33: BDD of example in Figure 6.26 using binary encoding of configuration variables and different orderings: all variables on top (a), arbitrary assignment (b), alternative arbitrary assignment(c), driven by architecture's topology (d).

Chapter 7

Experimental Evaluation of Exact Method

Having described the theoretical concepts underlying the automatic optimization approach in the previous chapters, in the following we describe the implementation and validation of our method, and present results to evaluate its performance.

7.1 Implementation Framework

The steps described in the previous section have been implemented in Python, exploiting the PySMT [294] library for SMT formulae manipulation and solving. PySMT is an open-source Python library that provides a solver agnostic interface to define, manipulate, and solve SMT formulae, leveraging the native *Application Programming Interface* (API)s of solvers or their SMTLIB [298] interface, allowing a fast prototyping of complex SMT-based algorithms. In particular, we used the following solvers:

- *MathSAT*: developed by FBK [299], it supports a wide range of theories and functionalities. We used it to quantify out non-Boolean variables through AllSMT.

- *CUDD*: developed by the Colorado University [300], it is a package for the manipulation of BDDs, ZBDDs, and other canonical representations of Boolean formulas. We used it to create and manipulate the OBDD representing the CSs of the architectures. Note: we used RePyCUDD, a Python wrapper for the CUDD BDD library.
- *Z3*: developed by Microsoft [301], it supports MOOP through SMT-based techniques. We employed it to explore the design space and find the design points.

PySMT interacts with each specific solver by means of a *converter* layer that converts each pySMT expression to a representation that the target solver can manage. We used this layer to access features that are exposed by the solver API but that are not wrapped by pySMT. This is the case of the All-SMT computation function that has to be called directly through the MathSAT python APIs by using a converter to create the internal MathSAT representation of an expression, and successively to reconstruct the PySMT version of the result.

7.2 Implementation Details

There are a few standard representations for architecture: a list of edges, an adjacency matrix, or an adjacency list. The choice of representation affects both the storage and computational time to perform look-ups and algorithms. The simplest way to implement that in Python is to use a "dictionary of lists", where each node n is a key with a list of neighbors, i.e. nodes connected to n . Hence, the basic system is represented as interconnected components, where each node corresponds to a basic component, and each edge denotes the flow of data between each component. For the creation and manipulation of system connections, we used NetworkX [302]. It is a Python package for the creation, manipulation, and study of the structure of complex networks, and management of data structures for graphs. We defined the logic of each block through SMT variables and constraints. We created a class that provides methods to generate the SMT formula representing all configurations and allow to operate on it in order to extract the reliability formula.

A library of candidate redundant patterns is associated to each node. Those patterns determine all the design alternatives that compose the design space. Afterwards, a formula modelling the CSA of each pair component-pattern is

created. Non-Boolean variables are then quantified out from each CSA by using the MathSAT function *All-SMT* that generates all the assignments to the Boolean variables that satisfy the formula, and that are used to compose an equisatisfiable formula in DNF.

As already stated before, the number of Boolean variables depends both on the number of sub-components of a pattern, which determine how many fault variables are used, and on the arity of each computing unit, which increases the number of abstract inputs. In order to optimize the overall computation, we exploited the analogies of the formulas modelling CSAs referring to the same pattern applied to components having same arity, defining the combinatorial abstract Miter presented in Section 6.3.10. Indeed, all these formulas can be cached and reused when needed, avoiding to perform every time the same (expensive) computation. PySMT offers APIs to serialize SMT formulae, but we opted for the more efficient python module named *Pickle*. It implements binary protocols for serializing and de-serializing a Python object structure. “Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy.

Once all Boolean formulae abstracting the behavior of each redundant component has been created, all the other constraints (i.e., linking constraints, compatibility constraints, probability constraints, and TLE formula) are extracted against the specification of basic system, and grouped together by putting a conjunction between them.

The abstracted inputs and outputs are then quantified out by simply converting the quantified PySMT formula to a CUDD structure through a converter. Since CUDD manipulates Boolean formulas through BDD structures, the tool implicitly eliminates the quantifiers of the PySMT formula through BDD-based techniques and creates the OBDD composed of fault and configuration variables.

BDDs are represented using complement edges, aka CCE BDD structures [303], allowing to minimize as much as possible the space required to encode the formula. CUDD framework handles all formulae through BDD structures, and it is therefore able to manage formulae consisting of Boolean variables only. For this reason, we explicitly binary encoded each configuration as a conjunction of Boolean variables without leveraging the SMT theory of bit-vectors, which would have made it possible to represent the configurations in a simpler and

more compact way. The conversion is illustrated in Figure 7.1. Note that, since we have assumed the hypothesis of monotonicity for our models, we can assert that they are also coherent. For coherent models, the definition of MCS fits well with the formal notion of *Prime Implicant* [304]. Once the CCE-BDD is created, we use the CUDD APIs to traverse the nodes of the OBDD, and create the SMT reliability formula as described in Paragraph 6.3.7.

The reliability assessment described above is the pillar of the DSE method proposed. In addition to reliability, other non-functional parameters were considered, such as cost, power consumption, and size. Since the focus of our work was the reliability, the other parameters were simply computed as additive, i.e. the cost of the system is calculated as the sum of the cost of single components. For the DSE we used the Z3 SMT optimizer through the PySMT APIs, because at the moment of writing it is the only one that supports the theory of *Non-Linear Arithmetic over the reals* ($\mathcal{NLR}\mathcal{A}$).

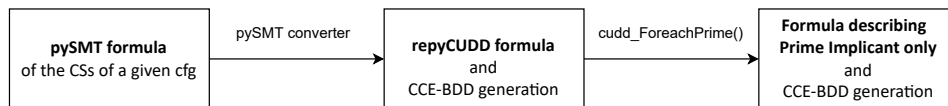


Figure 7.1: Conversion from PySMT formula to CCE-BDD representation using CUDD.

Summarizing what we have presented in the previous chapters, our software tool implements the following steps.

- Creating an SMT formula modelling the CSA of each pair component-pattern.
 - Using the basic version of the system as nominal behavior.
 - For each basic component, finding the applicable pattern with the highest number of fault variables for applying variable sharing: due to the mutual exclusivity, fault variables of different patterns associated to the same component can be reused.
- Creating the formula representing the CSs of the system.
 - Performing quantifier elimination via All-SMT on each CSA in order to obtain a formula with Boolean variables only.

- Caching the resulting formula in order to avoid the analog computation of a CSA related to the same pattern.
 - Linking the CSAs by adding additional Boolean constraints over input and output ports, according to the configuration variables (this translate into a conjunction of implications).
 - Adding compatibility constraints over configuration variables, to exclude the connection of patterns that are not sequentially compatible.
- Converting the formula of the CSs into a BDD representation.
 - Performing BDD-based quantifier elimination, in order to obtain a formula with only fault and configuration variables.
 - Apply the algorithm for the extraction of the symbolic formula of the reliability.
 - Performing the optimization on the basis of the reliability computed as specified above, and other non-functional parameters, whose computation may vary from case to case.

In appendix A the dependency graph of the software implementing our method is available.

All experiments have been executed on a desktop computer equipped with an Intel Core i5-3400 running Ubuntu 20.04.3.

7.3 Running Example

In the following, we firstly present some results on a simple case study, in order to show the impacts on results when varying the number of objective functions, and/or the number of redundant patterns available.

As a first example, we considered the system illustrated in Figure 7.2, which was also used by other authors [305], [154]. Although quite simple, it is a complex system and cannot therefore be broken down to groups of series and parallel components, which would make the reliability computation easier.

The system is composed of six components, each of which has two suitable redundant patterns, namely a TMR with one voter and a TMR with three

voters. The optimization problem involved a reliability function to be maximized and a cost function to be minimized. We assigned arbitrary values of fault probability and cost, as illustrated in Table 7.1.

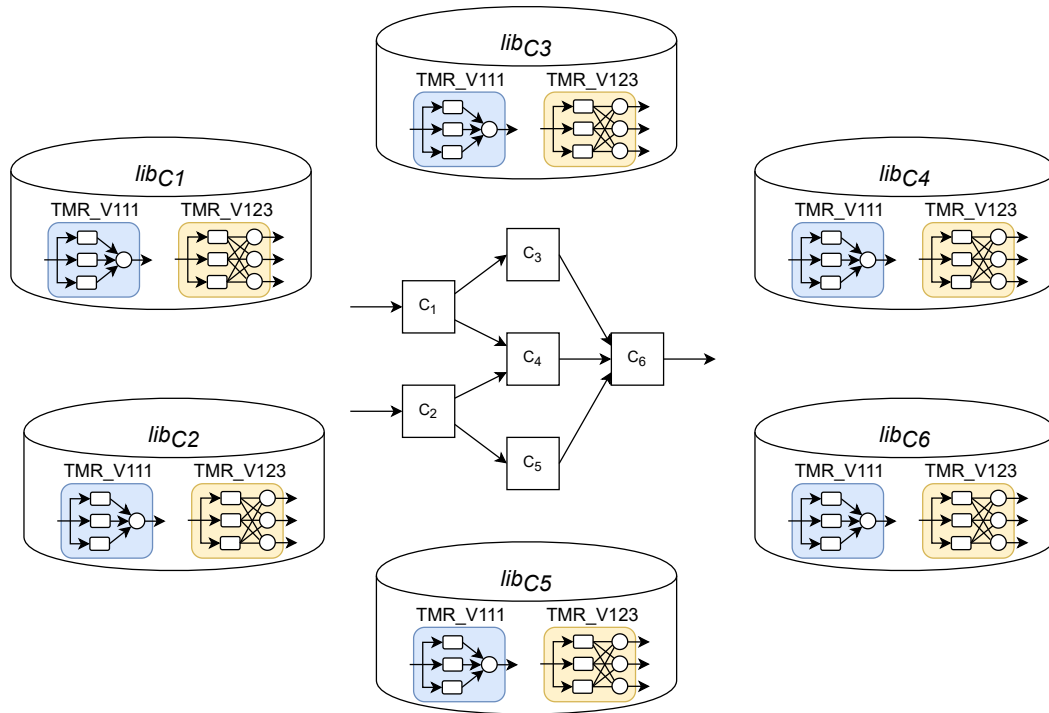


Figure 7.2: Basic system of first example.

Explicit Method

Our method, using explicit (aka enumerative) approach presented in Section 6.3.7, exceeded the timeout (set to twenty minutes) for the optimization task, although the basic system was made up of only six components. This happens because the number of alternative designs is combinatorial with respect to the number of components, and we need to extract the reliability separately for each one of them. What is even worse is that the time needed for reliability extraction grows very fast when the size of the system increases. This method can be very useful if we perform the reliability extraction of each alternative in parallel, using separate computing units.

Symbolic Method

Our method, using symbolic approach presented in Section 6.3.7, took 3.4 seconds for BDD quantifier elimination, but exceeded the timeout (set to twenty

Table 7.1: Library of patterns for example system in Figure 7.2, 2 objective functions.

Cmp	Ptn	Type	Pattern specification (Reliability, Cost)
C_1	P_1	<i>TMR_V111</i>	(0.2, 10), (0.2, 10), (0.2, 11), (0.1, 2)
	P_2	<i>TMR_V123</i>	(0.1, 10), (0.1, 10), (0.1, 10), (0.1, 2), (0.1, 2), (0.1, 2)
C_2	P_3	<i>TMR_V111</i>	(0.2, 11), (0.2, 11), (0.2, 11), (0.1, 4)
	P_4	<i>TMR_V123</i>	(0.1, 10), (0.1, 10), (0.1, 10), (0.1, 2), (0.1, 2), (0.1, 2)
C_3	P_5	<i>TMR_V111</i>	(0.2, 11), (0.2, 11), (0.2, 11), (0.1, 2)
	P_6	<i>TMR_V123</i>	(0.1, 10), (0.1, 10), (0.1, 10), (0.1, 2), (0.1, 2), (0.1, 2)
C_4	P_7	<i>TMR_V111</i>	(0.2, 11), (0.8, 11), (0.2, 11), (0.1, 2)
	P_8	<i>TMR_V123</i>	(0.1, 10), (0.1, 10), (0.1, 10), (0.1, 2), (0.1, 2), (0.1, 2)
C_5	P_9	<i>TMR_V111</i>	(0.2, 11), (0.2, 11), (0.2, 11), (0.2, 2)
	P_{10}	<i>TMR_V123</i>	(0.2, 11), (0.2, 11), (0.2, 11), (0.1, 2), (0.1, 2), (0.1, 2)
C_6	P_{11}	<i>TMR_V111</i>	(0.2, 12), (0.2, 12), (0.2, 12), (0.3, 3)
	P_{12}	<i>TMR_V123</i>	(0.1, 12), (0.1, 12), (0.1, 12), (0.1, 2), (0.1, 2), (0.1, 2)

minutes) for the optimization task. This is attributable to the high complexity of the symbolic representation.

Semi-symbolic Method

Our method, using semi-symbolic (aka Hybrid) approach presented in Section 6.3.7, produced eight solutions that define the best trade-off between the two competing objectives. The objective function values of these solutions are reported in Table 7.2. As could be expected, the result suggested that the most reliable solution is also the most expensive, and dually the cheapest is the one with highest value of fault probability. Figure 7.3 shows the Pareto set. Figure 7.4a shows one of the alternative solutions (number 6). Figure 7.4b shows the alternative solution with lower cost, Figure 7.4c shows the alternative solution with higher reliability.

Table 7.2: Exact solutions for example system in Figure 7.2, 2 objective functions.

Solution	Fault probability	Cost
1	0.673964	225
2	0.692971	222
3	0.705806	221
4	0.722956	218
5	0.746054	217
6	0.767727	215
7	0.782202	214
8	0.854541	213

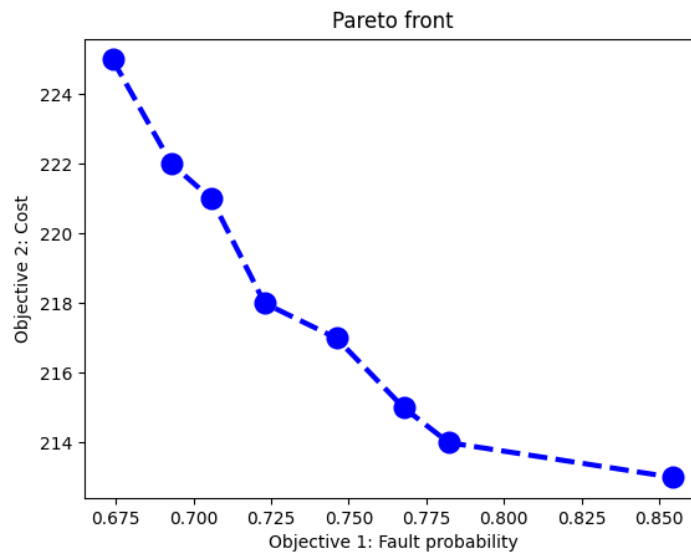


Figure 7.3: Pareto solutions for the example system in Figure 7.2

7.3.1 Varying the Number of Objective Functions

Usually, with MOOPs, we need to compare more than two objectives. To provide evidence of the efficiency of the proposed method, we added some others objectives. As next step, we considered one more objective function to be optimized (minimized): power consumption. We assigned arbitrary values as illustrated in Table 7.3. Our algorithm found 18 solutions, whose values are reported in Table 7.4. Figure 7.5 illustrates the Pareto surface of the three functions considered. The darker color corresponds to a higher fault probability. Let consider one more objective function: size area. Assigning arbitrary values as illustrated in Table 7.5, our algorithm found 21 solutions, whose values are reported in Table 7.6.

As additional step, we considered one more objective function to be optimized (minimized): weight. We assigned arbitrary values as illustrated in Table 7.7. Our algorithm found 28 solutions, whose values are reported in Table 7.8.

An additional objective function was the noise. Assigning arbitrary values as illustrated in Table 7.9, our algorithm found 40 solutions, whose values are reported in Table 7.10.

Figures 7.6 and 7.7 show the performance of our algorithm when varying the number of objective functions. Please note that time is higher with two

Table 7.3: Library of patterns for example system in Figure 7.2, considering 3 objective functions.

Ptn	Pattern specification (Reliability, Cost, Power)
P_1	(0.2, 10, 9), (0.2, 10, 9), (0.2, 11, 9), (0.1, 2, 3)
P_2	(0.1, 10, 11), (0.1, 10, 11), (0.1, 10, 11), (0.1, 2, 4), (0.1, 2, 4), (0.1, 2, 4)
P_3	(0.2, 11, 13), (0.2, 11, 13), (0.2, 11, 13), (0.1, 4, 2)
P_4	(0.1, 10, 12), (0.1, 10, 12), (0.1, 10, 12), (0.1, 2, 3), (0.1, 2, 3), (0.1, 2, 3)
P_5	(0.2, 11, 15), (0.2, 11, 15), (0.2, 11, 15), (0.1, 2, 3)
P_6	(0.1, 10, 12), (0.1, 10, 12), (0.1, 10, 12), (0.1, 2, 2), (0.1, 2, 2), (0.1, 2, 2)
P_7	(0.2, 11, 12), (0.8, 11, 12), (0.2, 11, 12), (0.1, 2, 2)
P_8	(0.1, 10, 14), (0.1, 10, 14), (0.1, 10, 14), (0.1, 2, 3), (0.1, 2, 3), (0.1, 2, 3)
P_9	(0.2, 11, 12), (0.2, 11, 12), (0.2, 11, 12), (0.2, 2, 5)
P_{10}	(0.2, 11, 13), (0.2, 11, 13), (0.2, 11, 13), (0.1, 2, 4), (0.1, 2, 4), (0.1, 2, 4)
P_{11}	(0.2, 12, 11), (0.2, 12, 11), (0.2, 12, 11), (0.3, 3, 4)
P_{12}	(0.1, 12, 12), (0.1, 12, 12), (0.1, 12, 12), (0.1, 2, 3), (0.1, 2, 3), (0.1, 2, 3)

Table 7.4: Solutions for example system in Figure 7.2, considering 3 objective functions.

Solution	Fault probability	Cost	Power
1	0.6739644436312434	225.0	279.0
2	0.6844985760212222	226.0	275.0
3	0.6929706872988478	222.0	264.0
4	0.7058063295659203	221.0	269.0
5	0.7084385079107389	223.0	260.0
6	0.7126954744932213	221.0	261.0
7	0.7229563504041777	218.0	254.0
8	0.7369135241864181	219.0	250.0
9	0.7460535490548722	217.0	251.0
10	0.7588471097533224	218.0	247.0
11	0.7677269176540538	215.0	246.0
12	0.7794285963605267	216.0	242.0
13	0.7822023816175623	214.0	243.0
14	0.7931748013555948	215.0	239.0
15	0.8374944867793959	215.0	238.0
16	0.8417515885357502	214.0	242.0
17	0.8506274578629505	214.0	235.0
18	0.8545405196347040	213.0	239.0

functions because subsequent cases take advantage of the caching mechanism, i.e., for some patterns they use formulae already computed and stored. A qualitative method for the visualization of the Pareto set in the objective space that aims at identifying the broad trends is illustrated in Figure 7.8, through parallel coordinate plots. Each objective is given its own axis and all

Table 7.5: Library of patterns for example system in Figure 7.2, considering 4 objective functions.

Ptn	Pattern specification (Reliability, Cost, Power, Size)
P_1	(0.2, 10, 9, 16), (0.2, 10, 9, 16), (0.2, 11, 9, 16), (0.1, 2, 3, 3)
P_2	(0.1, 10, 11, 20), (0.1, 10, 11, 20), (0.1, 10, 11, 20), (0.1, 2, 4, 3), (0.1, 2, 4, 3), (0.1, 2, 4, 3)
P_3	(0.2, 11, 13, 17), (0.2, 11, 13, 17), (0.2, 11, 13, 17), (0.1, 4, 2, 3)
P_4	(0.1, 10, 12, 22), (0.1, 10, 12, 22), (0.1, 10, 12, 22), (0.1, 2, 3, 2), (0.1, 2, 3, 2), (0.1, 2, 3, 2)
P_5	(0.2, 11, 15, 16), (0.2, 11, 15, 16), (0.2, 11, 15, 16), (0.1, 2, 3, 3)
P_6	(0.1, 10, 12, 21), (0.1, 10, 12, 21), (0.1, 10, 12, 21), (0.1, 2, 2, 4), (0.1, 2, 2, 4), (0.1, 2, 2, 4)
P_7	(0.2, 11, 12, 15), (0.8, 11, 12, 15), (0.2, 11, 12, 15), (0.1, 2, 2, 5)
P_8	(0.1, 10, 14, 19), (0.1, 10, 14, 19), (0.1, 10, 14, 19), (0.1, 2, 3, 4), (0.1, 2, 3, 4), (0.1, 2, 3, 4)
P_9	(0.2, 11, 12, 16), (0.2, 11, 12, 16), (0.2, 11, 12, 16), (0.2, 2, 5, 4)
P_{10}	(0.2, 11, 13, 21), (0.2, 11, 13, 21), (0.2, 11, 13, 21), (0.1, 2, 4, 3), (0.1, 2, 4, 3), (0.1, 2, 4, 3)
P_{11}	(0.2, 12, 11, 14), (0.2, 12, 11, 14), (0.2, 12, 11, 14), (0.3, 3, 4, 3)
P_{12}	(0.1, 12, 12, 18), (0.1, 12, 12, 18), (0.1, 12, 12, 18), (0.1, 2, 3, 2), (0.1, 2, 3, 2), (0.1, 2, 3, 2)

Table 7.6: Solutions for example system in Figure 7.2, considering 4 objective functions.

Solution	Fault probability	Cost	Power	Size
1	0.6739644436312434	225.0	279.0	417.0
2	0.6844985760212222	226.0	275.0	399.0
3	0.6929706872988478	222.0	264.0	399.0
4	0.702429368641965,	224.0	276.0	393.0
5	0.7058063295659203	221.0	269.0	397.0
6	0.7084385079107389	223.0	260.0	381.0
7	0.7126954744932213	221.0	261.0	375.0
8	0.7229563504041777	218.0	254.0	379.0
9	0.7271695806377669	222.0	257.0	357.0
10	0.7369135241864181	219.0	250.0	361.0
11	0.7460535490548722	217.0	251.0	355.0
12	0.7588471097533224	218.0	247.0	337.0
13	0.7677269176540538	215.0	246.0	364.0
14	0.7794285963605267	216.0	242.0	346.0
15	0.7822023816175623	214.0	243.0	340.0
16	0.7931748013555948	215.0	239.0	322.0
17	0.8312442747324496	217.0	243.0	318.0
18	0.8374944867793959	215.0	238.0	327.0
19	0.8417515885357502	214.0	242.0	345.0
20	0.8506274578629505	214.0	235.0	303.0
21	0.8545405196347040	213.0	239.0	321.0

the axes are placed in parallel to each other. Values are plotted as a series of lines that connect across all the axes. They are ideal for comparing many variables together and seeing the relationships between them. As we can see, Cost, Power, Size, and Weight are inversely proportional to Fault probability.

Table 7.7: Library of patterns for example system in Figure 7.2, considering 5 objective functions.

Ptn	Pattern specification (Reliability, Cost, Power, Size, Weight)
P_1	(0.2, 10, 9, 16, 2.7), (0.2, 10, 9, 16, 2.7), (0.2, 11, 9, 16, 2.7), (0.1, 2, 3, 3, 2.7)
P_2	(0.1, 10, 11, 20, 2.5), (0.1, 10, 11, 20, 2.5), (0.1, 10, 11, 20, 2.5), (0.1, 2, 4, 3, 2.5), (0.1, 2, 4, 3, 2.5), (0.1, 2, 4, 3, 2.5)
P_3	(0.2, 11, 13, 17, 2), (0.2, 11, 13, 17, 2), (0.2, 11, 13, 17, 2), (0.1, 4, 2, 3, 2.1)
P_4	(0.1, 10, 12, 22, 2.1), (0.1, 10, 12, 22, 2.1), (0.1, 10, 12, 22, 2.1), (0.1, 2, 3, 2, 2.1), (0.1, 2, 3, 2, 2.1), (0.1, 2, 3, 2, 2.1)
P_5	(0.2, 11, 15, 16, 2.7), (0.2, 11, 15, 16, 2.7), (0.2, 11, 15, 16, 2.7), (0.1, 2, 3, 3, 2.6)
P_6	(0.1, 10, 12, 21, 2.2), (0.1, 10, 12, 21, 2.2), (0.1, 10, 12, 21, 2.2), (0.1, 2, 2, 4, 2.1), (0.1, 2, 2, 4, 2.1), (0.1, 2, 2, 4, 2.1)
P_7	(0.2, 11, 12, 15, 3.3), (0.8, 11, 12, 15, 3.3), (0.2, 11, 12, 15, 3.3), (0.1, 2, 2, 5, 3.4)
P_8	(0.1, 10, 14, 19, 3.1), (0.1, 10, 14, 19, 3.1), (0.1, 10, 14, 19, 3.1), (0.1, 2, 3, 4, 3.2), (0.1, 2, 3, 4, 3.2), (0.1, 2, 3, 4, 3.2)
P_9	(0.2, 11, 12, 16, 1.8), (0.2, 11, 12, 16, 1.8), (0.2, 11, 12, 16, 1.8), (0.2, 2, 5, 4, 1.5)
P_{10}	(0.2, 11, 13, 21, 2.6), (0.2, 11, 13, 21, 2.6), (0.2, 11, 13, 21, 2.6), (0.1, 2, 4, 3, 2.1), (0.1, 2, 4, 3, 2.1), (0.1, 2, 4, 3, 2.1)
P_{11}	(0.2, 12, 11, 14, 5), (0.2, 12, 11, 14, 5), (0.2, 12, 11, 14, 5), (0.3, 3, 4, 3, 1.5)
P_{12}	(0.1, 12, 12, 18, 4.9), (0.1, 12, 12, 18, 4.9), (0.1, 12, 12, 18, 4.9), (0.1, 2, 3, 2, 2.9), (0.1, 2, 3, 2, 2.9), (0.1, 2, 3, 2, 2.9)

Table 7.8: Solutions for example system in Figure 7.2, considering 5 objective functions.

Solution	Fault probability	Cost	Power	Size	Weight
1	0.6739644436312434	225.0	279.0	417.0	282.030
2	0.6844985760212222	226.0	275.0	399.0	271.980
3	0.6929706872988478	222.0	264.0	399.0	273.690
4	0.7024293686419650	224.0	276.0	393.0	282.910
5	0.7058063295659203	221.0	269.0	397.0	260.490
6	0.7084385079107389	223.0	260.0	381.0	263.640
7	0.7126954744932213	225.0	272.0	375.0	272.860
8	0.7126954744932213	221.0	261.0	375.0	274.570
9	0.7153116580864123	222.0	265.0	379.0	250.440
10	0.7229563504041777	218.0	254.0	379.0	252.150
11	0.7271695806377669	222.0	257.0	357.0	264.520
12	0.7369135241864181	219.0	250.0	361.0	242.100
13	0.7460535490548722	221.0	262.0	355.0	251.320
14	0.7460535490548722	217.0	251.0	355.0	253.030
15	0.7533483595885260	218.0	261.0	382.0	240.480
16	0.7588471097533224	218.0	247.0	337.0	242.980
17	0.7613176162648169	219.0	257.0	364.0	230.430
18	0.7677269176540538	215.0	246.0	364.0	232.140
19	0.7744199305735989	217.0	258.0	358.0	241.360
20	0.7794285963605267	216.0	242.0	346.0	222.090
21	0.7822023816175623	218.0	254.0	340.0	231.310
22	0.7822023816175623	214.0	243.0	340.0	233.020
23	0.7931748013555948	215.0	239.0	322.0	222.970
24	0.8312442747324496	217.0	243.0	318.0	227.660
25	0.8374944867793959	215.0	238.0	327.0	206.770
26	0.8417515885357502	214.0	242.0	345.0	216.820
27	0.8506274578629505	214.0	235.0	303.0	207.650
28	0.8545405196347040	213.0	239.0	321.0	217.700

Table 7.9: Library of patterns for example system in Figure 7.2, considering 6 objective functions.

Ptn	Pattern specification (Reliability, Cost, Power, Size, Weight, Noise)
P_1	(0.2, 10, 9, 16, 2.7, 8), (0.2, 10, 9, 16, 2.7, 8), (0.2, 11, 9, 16, 2.7, 8), (0.1, 2, 3, 3, 2.7, 3)
P_2	(0.1, 10, 11, 20, 2.5, 7), (0.1, 10, 11, 20, 2.5, 7), (0.1, 10, 11, 20, 2.5, 7), (0.1, 2, 4, 3, 2.5, 2), (0.1, 2, 4, 3, 2.5, 2), (0.1, 2, 4, 3, 2.5, 2)
P_3	(0.2, 11, 13, 17, 2, 10), (0.2, 11, 13, 17, 2, 10), (0.2, 11, 13, 17, 2, 10), (0.1, 4, 2, 3, 2.1, 4)
P_4	(0.1, 10, 12, 22, 2.1, 9), (0.1, 10, 12, 22, 2.1, 9), (0.1, 10, 12, 22, 2.1, 9), (0.1, 2, 3, 2, 2.1, 3), (0.1, 2, 3, 2, 2.1, 3), (0.1, 2, 3, 2, 2.1, 3)
P_5	(0.2, 11, 15, 16, 2.7, 6), (0.2, 11, 15, 16, 2.7, 6), (0.2, 11, 15, 16, 2.7, 6), (0.1, 2, 3, 3, 2.6, 4)
P_6	(0.1, 10, 12, 21, 2.2, 5), (0.1, 10, 12, 21, 2.2, 5), (0.1, 10, 12, 21, 2.2, 5), (0.1, 2, 2, 4, 2.1, 2), (0.1, 2, 2, 4, 2.1, 2)
P_7	(0.2, 11, 12, 15, 3.3, 7), (0.8, 11, 12, 15, 3.3, 7), (0.2, 11, 12, 15, 3.3, 7), (0.1, 2, 2, 5, 3.4, 4)
P_8	(0.1, 10, 14, 19, 3.1, 4), (0.1, 10, 14, 19, 3.1, 4), (0.1, 10, 14, 19, 3.1, 4), (0.1, 2, 3, 4, 3.2, 4), (0.1, 2, 3, 4, 3.2, 4), (0.1, 2, 3, 4, 3.2, 4)
P_9	(0.2, 11, 12, 16, 1.8, 8), (0.2, 11, 12, 16, 1.8, 8), (0.2, 11, 12, 16, 1.8, 8), (0.2, 2, 5, 4, 1.5, 3)
P_{10}	(0.2, 11, 13, 21, 2.6, 7), (0.2, 11, 13, 21, 2.6, 7), (0.2, 11, 13, 21, 2.6, 7), (0.1, 2, 4, 3, 2.1, 3), (0.1, 2, 4, 3, 2.1, 3), (0.1, 2, 4, 3, 2.1, 3)
P_{11}	(0.2, 12, 11, 14, 5, 9), (0.2, 12, 11, 14, 5, 9), (0.2, 12, 11, 14, 5, 9), (0.3, 3, 4, 3, 1.5, 2)
P_{12}	(0.1, 12, 12, 18, 4.9, 6), (0.1, 12, 12, 18, 4.9, 6), (0.1, 12, 12, 18, 4.9, 6), (0.1, 2, 3, 2, 2.9, 5), (0.1, 2, 3, 2, 2.9, 5), (0.1, 2, 3, 2, 2.9, 5)

Table 7.10: Solutions for example system in Figure 7.2, considering 6 objective functions.

Solution	Fault probability	Cost	Power	Size	Weight	Noise
1	0.7460535490548722	217.0	251.0	355.0	253.030	7513.0
2	0.6739644436312434	225.0	279.0	417.0	282.030	6237.0
3	0.6844985760212222	226.0	275.0	399.0	271.980	7033.0
4	0.6929706872988478	222.0	264.0	399.0	273.690	6747.0
5	0.7024293686419650	224.0	276.0	393.0	282.910	6550.0
6	0.7058063295659203	221.0	269.0	397.0	260.490	6690.0
7	0.7084385079107389	223.0	260.0	381.0	263.640	7543.0
8	0.7126954744932213	225.0	272.0	375.0	272.860	7346.0
9	0.7126954744932213	221.0	261.0	375.0	274.570	7060.0
10	0.7153116580864123	222.0	265.0	379.0	250.440	7486.0
11	0.7229563504041777	218.0	254.0	379.0	252.150	7200.0
12	0.7271695806377669	222.0	257.0	357.0	264.520	7856.0
13	0.7369135241864181	219.0	250.0	361.0	242.100	7996.0
14	0.7369794102422135	220.0	266.0	373.0	261.370	7003.0
15	0.7460535490548722	221.0	262.0	355.0	251.320	7799.0
16	0.7533483595885260	218.0	261.0	382.0	240.480	7862.0
17	0.7588471097533224	218.0	247.0	337.0	242.980	8309.0
18	0.7613176162648169	219.0	257.0	364.0	230.430	8658.0
19	0.7677269176540538	215.0	246.0	364.0	232.140	8372.0
20	0.7744199305735989	217.0	258.0	358.0	241.360	8175.0
21	0.7794285963605267	216.0	242.0	346.0	222.090	9168.0
22	0.7822023816175623	218.0	254.0	340.0	231.310	8971.0
23	0.7822023816175623	214.0	243.0	340.0	233.020	8685.0
24	0.7931748013555948	215.0	239.0	322.0	222.970	9481.0
25	0.8105227290918962	218.0	246.0	342.0	226.780	8705.0
26	0.8124832497987964	221.0	261.0	360.0	235.120	8195.0
27	0.8154864008270026	217.0	250.0	360.0	236.830	7909.0
28	0.8179286704985339	220.0	265.0	378.0	245.170	7399.0
29	0.8312442747324496	217.0	243.0	318.0	227.660	9018.0
30	0.8356651111717405	216.0	247.0	336.0	237.709	8222.0
31	0.8356651111717405	220.0	258.0	336.0	236.000	8508.0
32	0.8374944867793959	215.0	238.0	327.0	206.770	9877.0
33	0.8391759307970742	218.0	253.0	345.0	215.110	9367.0
34	0.8403771134483601	219.0	262.0	354.0	246.050	7712.0
35	0.8417515885357502	214.0	242.0	345.0	216.820	9081.0
36	0.8438462053966178	217.0	257.0	363.0	225.160	8571.0
37	0.8506274578629505	214.0	235.0	303.0	207.650	10190.0
38	0.8545405196347040	217.0	250.0	321.0	215.990	9680.0
39	0.8545405196347040	213.0	239.0	321.0	217.700	9394.0
40	0.8587113041073394	216.0	254.0	339.0	226.040	8884.0

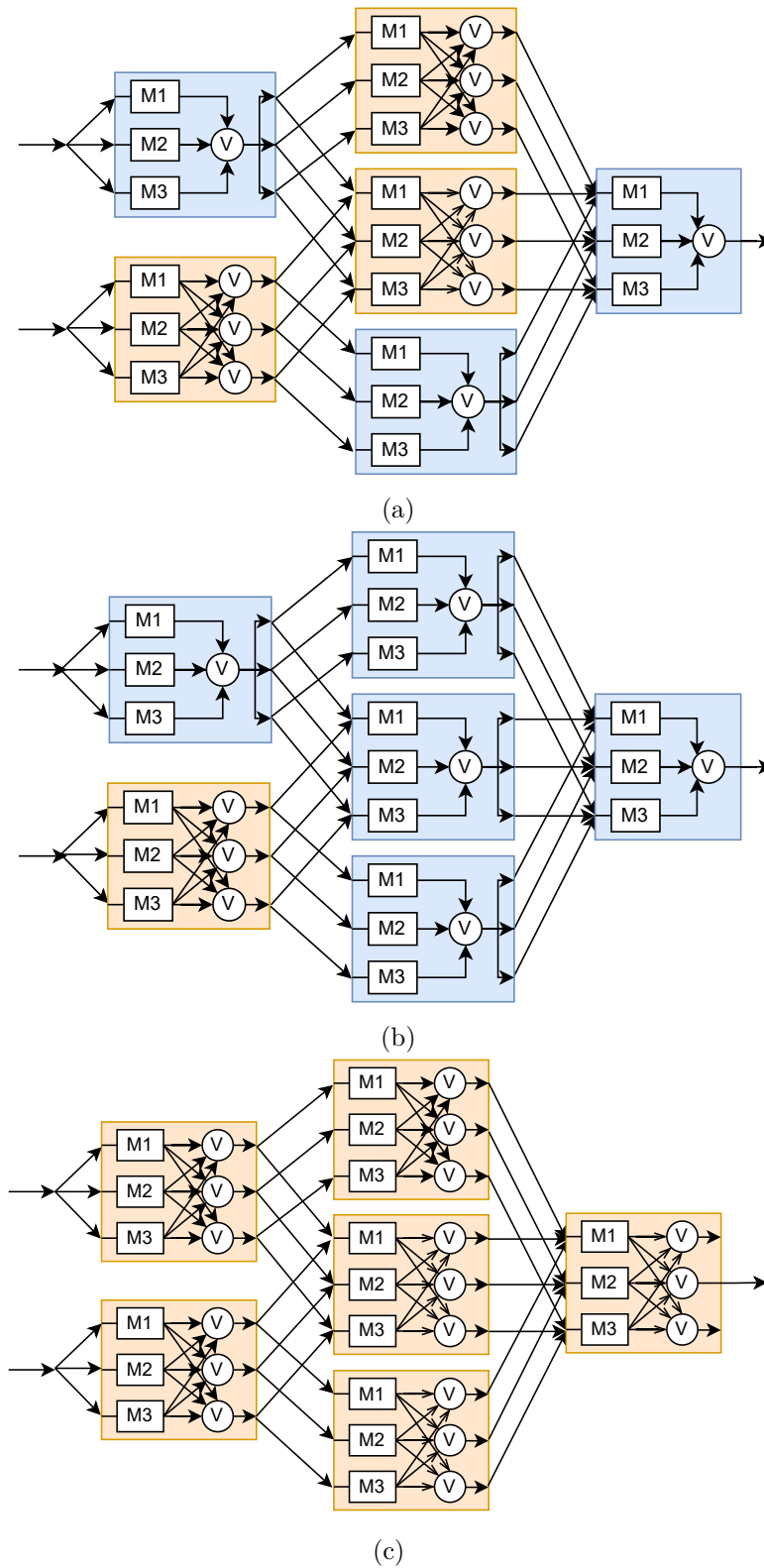


Figure 7.4: Alternative solutions for example system in Figure 7.2: (a) solution n.6, (b) redundant architecture scheme with minimum cost, and (c) redundant architecture scheme with maximum reliability.

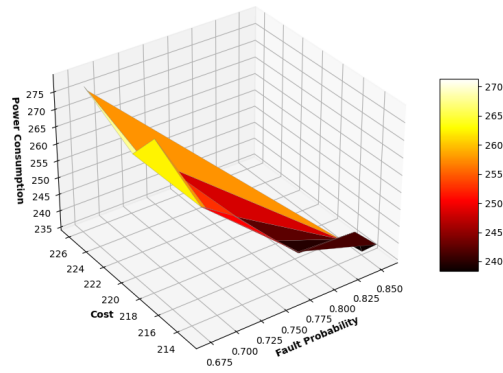


Figure 7.5: Pareto surface for example system in Figure 7.2 with three objective functions.

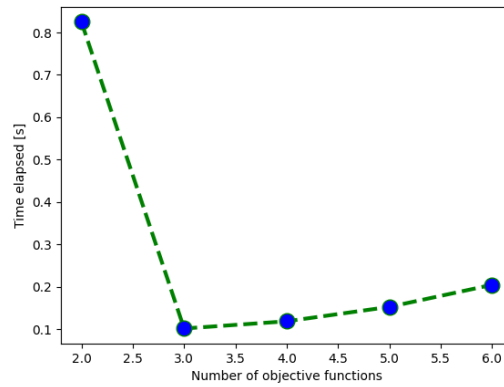


Figure 7.6: Time performance for optimization when varying the number of objectives for example system in Figure 7.2

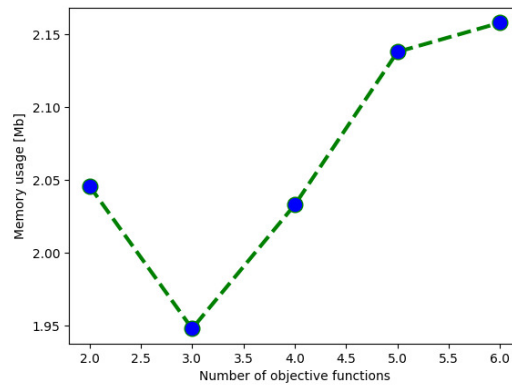


Figure 7.7: Memory usage for optimization when varying the number of objectives for example system in Figure 7.2

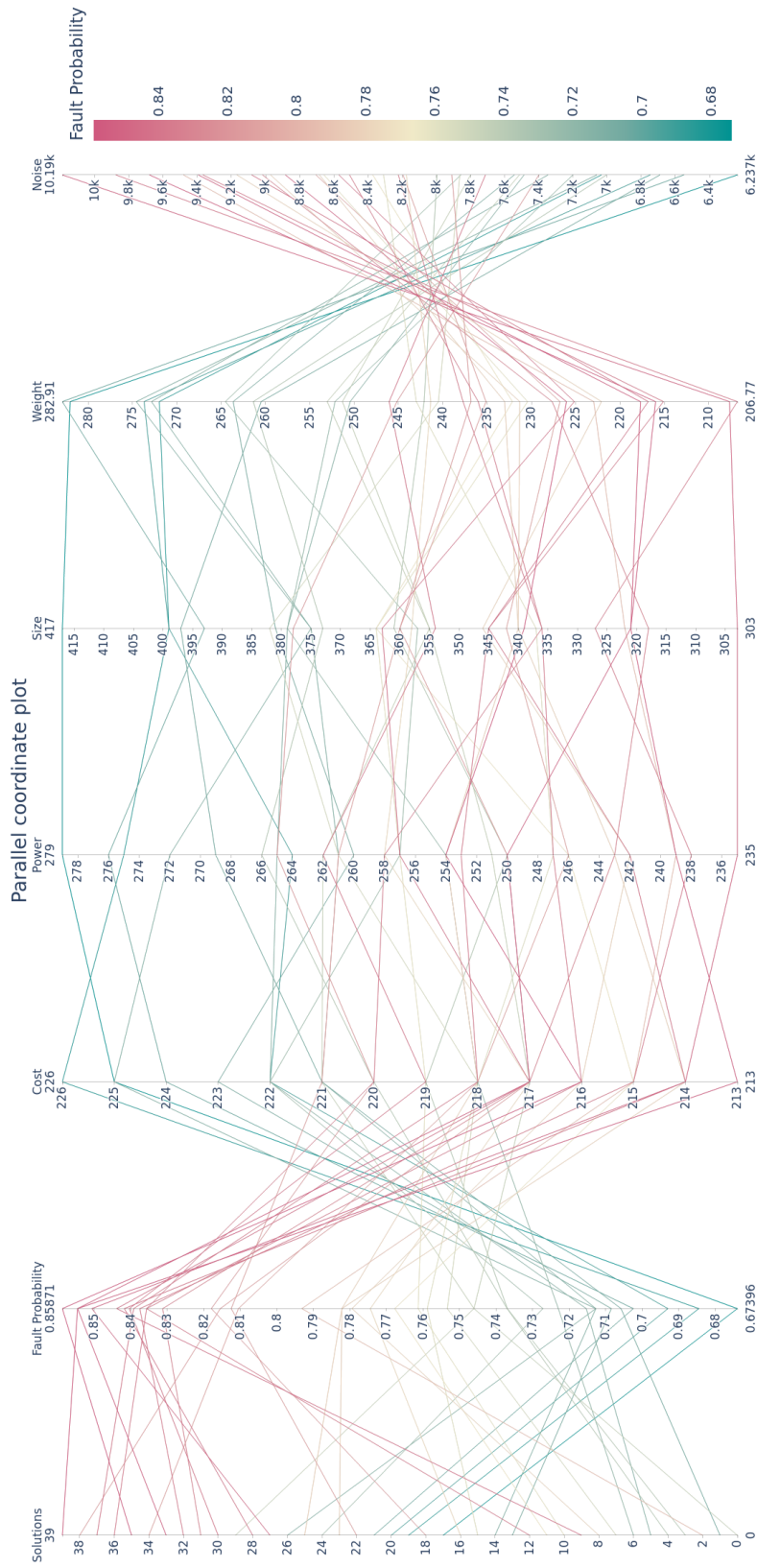


Figure 7.8: Parallel coordinate plot for the running example with six objective functions

7.3.2 Varying the Number of Redundant Patterns

In order to test the performance of the algorithm proposed, we can vary the size of the library of redundant patterns. Specifically, to the initial library composed of only two patterns for each component we add in turn up to five patterns, namely: *TMR_V111*, *TMR_V123*, *TMR_V001*, *TMR_V010*, and *TMR_V012*. The same library can include two or more instances of the same pattern type. Table 7.11 reports the performance of the proposed method and the number of solutions found by varying the size of the library and the number of objective functions. The timeout for computation was set to twenty minutes.

Table 7.11: Time and memory performance for the complex system of six basic components illustrated in Figure 7.2

Library size	Objectives	Time elapsed [s]	Memory usage [Mb]	Number of solutions
2	2	0.825	2.046	8
	3	0.102	1.948	18
	4	0.119	2.033	21
	5	0.153	2.138	28
	6	0.205	2.158	40
3	2	1.429	2.202	23
	3	2.638	1.880	99
	4	118.01	2.015	154
	5	timeout	-	-
	6	timeout	-	-
4	2	43.535	2.225	23
	3	53.291	1.938	166
	4	timeout	-	-
	5	timeout	-	-
	6	timeout	-	-

7.3.3 Varying the Number of System Components

Varying the number of components means considering new systems. A deepened analysis that considers - among others - different number of components is performed in the next section.

7.3.4 Varying the BDD Ordering Strategy

Among other things, we also investigated how the BDD ordering influences its size and, as a consequence, the performance of our method. The size of a BDD for a given function is sensitive to the ordering of the variable in the BDD. However, finding the optimal ordering that yields the smallest BDD for the

given function is an NP-complete problem [306]. Several heuristic algorithms have been therefore developed in order to find a variable ordering that help reducing the BDD size.

The CUDD package that we have employed includes an implementation of the SIFT algorithm [297], which is one of the most popular ones based on *dynamic variable reordering* [307]. SIFT algorithm picks one BDD variable at a time and tries to find a better position for this variable, such that the resulting BDD is smaller in size. This algorithm is efficient because the search for a better position is based on the exchange of adjacent variables, which turns out to be an efficient operation. Unfortunately, the algorithm does not provide guarantee to find the best ordering, or can even run out of memory. For this reason, as already presented in Sub-section 6.3.10, we considered the following orderings:

- a static ordering in which all configuration variables precede fault variables,
- a static ordering driven by architecture,
- an arbitrary ordering.

Through the APIs of CUDD, it is possible to select a large variety of reordering strategies for the last case. Since the BDDs under study have a high number of variables, we opted for the SIFT algorithm, which is agile. Other algorithms like GA-based algorithms take long times even with very simple systems. We applied the SIFT algorithm to the running example, but since with complex architectures the reordering strategy adds a considerable overhead, the execution of the experiment reached the timeout. More details and numerical results are provided in the next sections.

7.4 Benchmarks

As we have done for the example provided in the previous section, in order to verify the feasibility and scalability of our method, we considered different test scenarios, by varying:

- the kind of architecture (see Figure 7.9),
- the number of components,

- the number and type of patterns available for each component,
- the number of instances of the same pattern,
- the number of objective functions,
- the kind of optimization approach: explicit, symbolic, or semi-symbolic (aka hybrid).

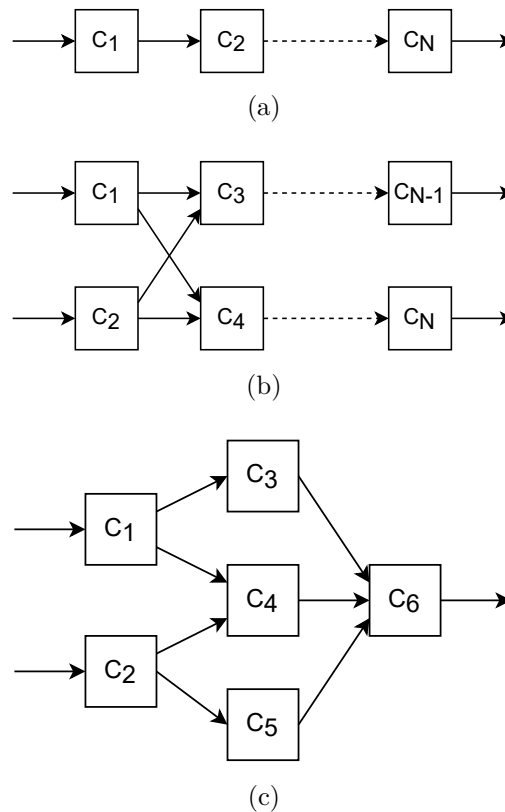


Figure 7.9: System architectures used for experimental evaluation: (a) series (aka linear) (b) repeating pairs of parallel components (aka rectangular), (c) complex random architectures.

7.4.1 Experimental Setup

The setting for the experimental evaluation comprises the three steps illustrated in Figure 6.1. This means the generation of the basic architecture, expressed in terms of nodes and connections between them, the generation of the pattern library for each component, the definition of the configuration

variables and fault variables, and the subsequent generation of the abstract modules. Afterward, the next step is the construction of the combinatorial abstract Miter and the computation of its MCS. The formula of the MCS is then translated into a BDD that is traversed in order to evaluate the reliability of the redundant systems. The evaluation of the other objectives is translated into a simple addition operation. Eventually, a solver is used to perform the optimization task.

7.4.2 Evaluation Criteria

As stated above, we considered different kinds of architectures, specifically series (aka linear), repeating pairs of parallel components (aka rectangular), and complex redundant architectures, varying the number of basic components, patterns, instances of the same pattern, and design objectives. The overall performance has two main contributions: the assessment of non-functional parameters and optimization. In order to get a rough measurement of the time complexity of the algorithm, the overall CPU time and memory usage are measured when running the experiments. Another evaluation parameter is the number of nodes of the BDDs generated, which can help figuring out the dimension and complexity of the system we are dealing with.

7.4.3 Experiments on Linear Architectures

The basic architecture consists of a chain of basic components. We varied the number of basic components up to one-hundred, the number of redundant patterns up to ten for each basic component, the number of different instances of the same component up to five, and the number of objectives up to six. As expected, the results indicated that the time for the BDD-based quantifier elimination grows with the number of components and with the size of the pattern library, as illustrated in Figure 7.10. Thus, the performance is proportional to the size of the BDD. Additional information is provided in Section 7.5 below.

As predictable, also the time needed to traverse the BDD and extract the reliability formula depends on the size of the BDD. It is linear with respect to the size of the BDD.

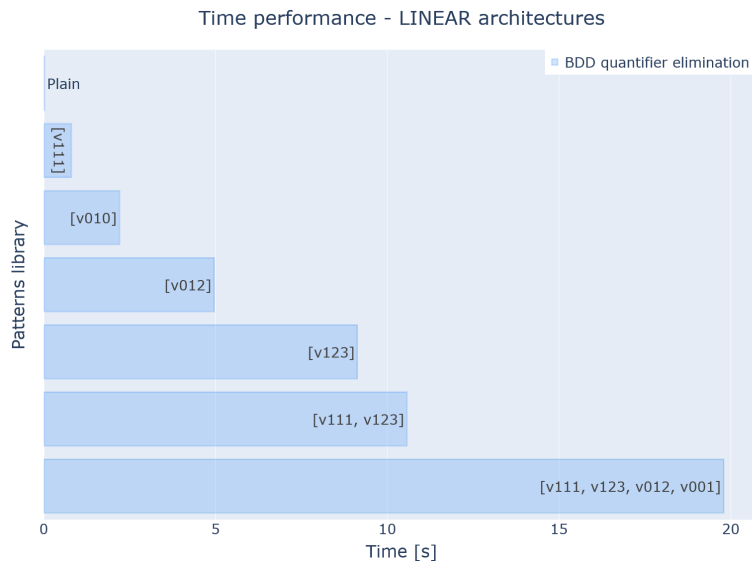


Figure 7.10: Time performance for linear architectures, varying the size of the system and library of patterns.

7.4.4 Experiments on Rectangular Architectures

As per linear architectures, also for rectangular architectures the time for the BDD-based quantifier elimination and reliability function extraction grows with the number of components and with the size of the pattern library, as illustrated in Figure 7.11.

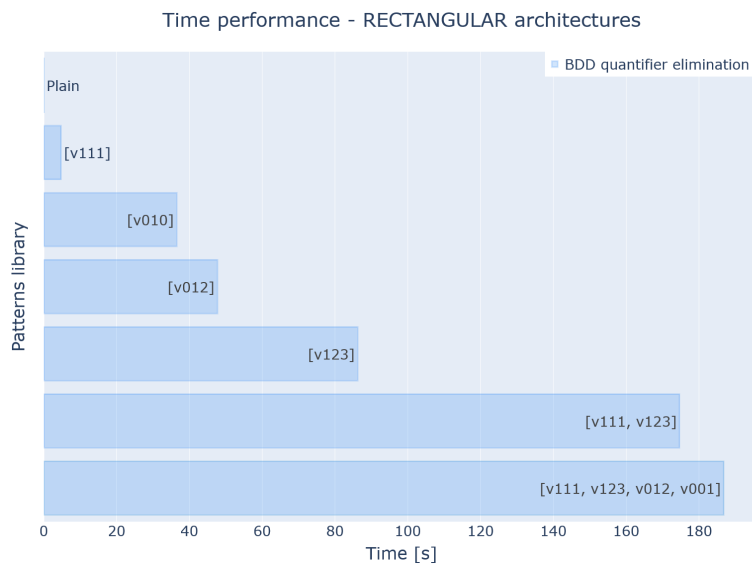


Figure 7.11: Time performance for rectangular architectures, varying the size of the system and library of patterns.

7.4.5 Experiments on Complex Architectures

The topology is modeled by a randomly generated graph. This graph is a DAG in which each node has a maximum input degree d (i.e., each node has maximum d incoming edges). In the following we report the outcomes of some of the experiments we have performed. In order to illustrate how the number of solutions and the performance change with the number of basic components, we kept fixed the size of the pattern library (three patterns per each component) and the number of objective functions (two objectives: maximize reliability and minimize cost), while we varied the number of the components (and edges). Table 7.12 and Figure 7.16 illustrate the outcomes. Compared to linear and rectangular architectures, the results indicated that curves for complex architectures become steeper already with few components (less than ten). Especially if using more than five patterns for each of them. More details follow in the next section.

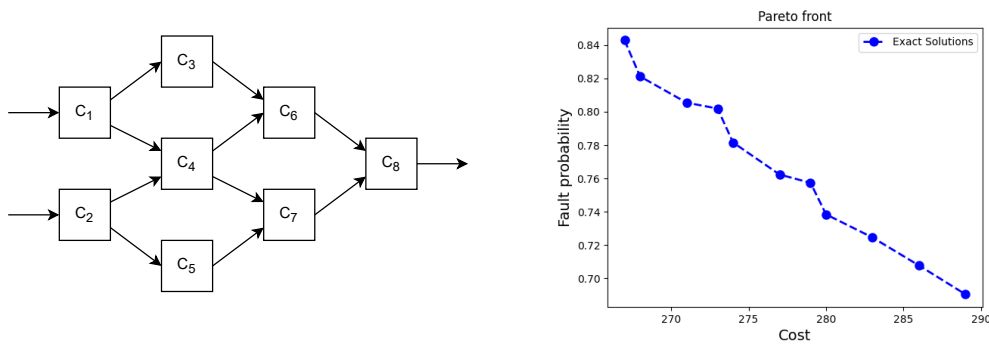


Figure 7.12: Pareto solutions for a basic system composed of 8 components and 10 edges.

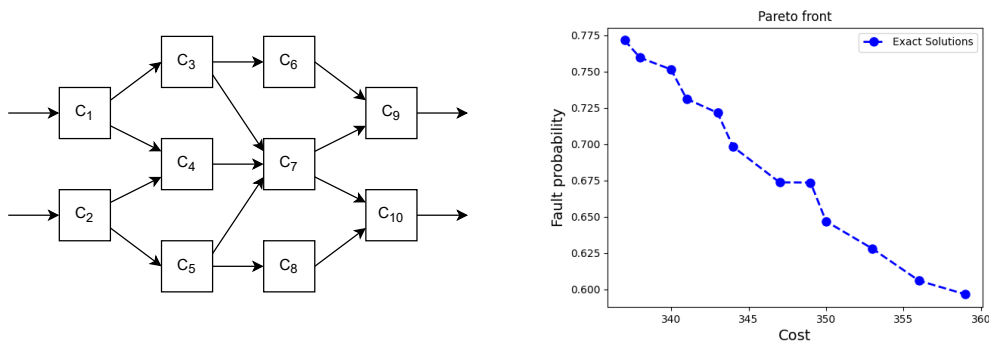


Figure 7.13: Pareto solutions for a basic system composed of 10 components and 10 edges.

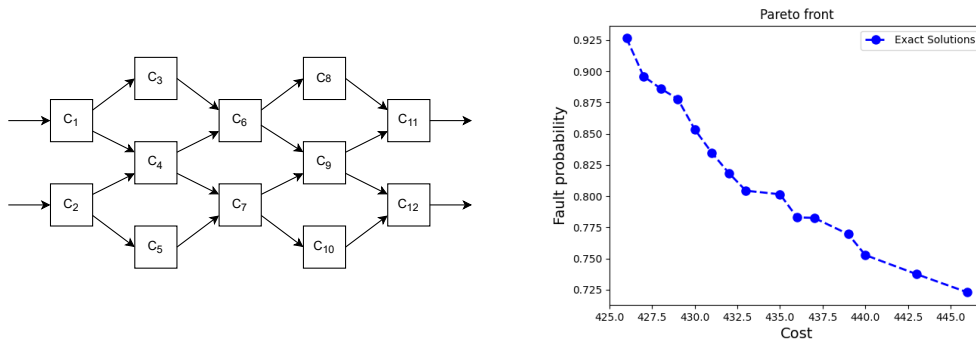


Figure 7.14: Pareto solutions for a basic system composed of 12 components and 16 edges.

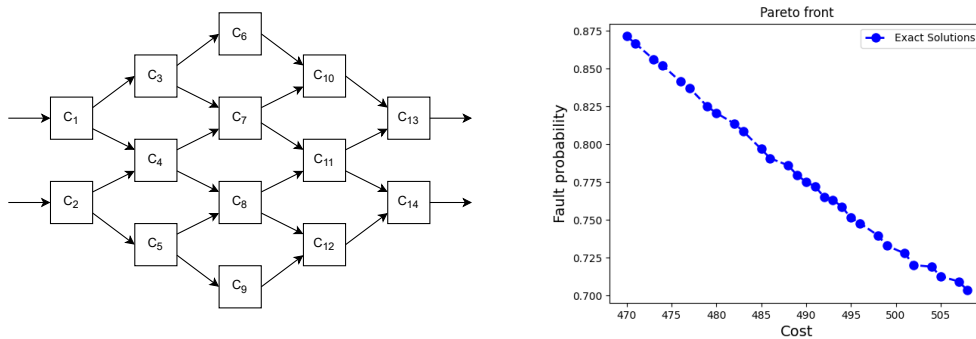


Figure 7.15: Pareto solutions for a basic system composed of 14 components and 20 edges.

7.5 Results

As stated above, the overall performance of the exact method has two main contributions: assessment of non-functional parameters and optimization. We present the respective results in the following.

Table 7.12: Performance for the complex system examples presented above.

Number of components	6	8	10	12	14
Number of solutions	8	11	12	15	29
Time elapsed [s]	24.38	36.71	213.27	2240.51	15630.07
Memory usage [Mb]	2.11	2.53	3.09	3.07	3.41

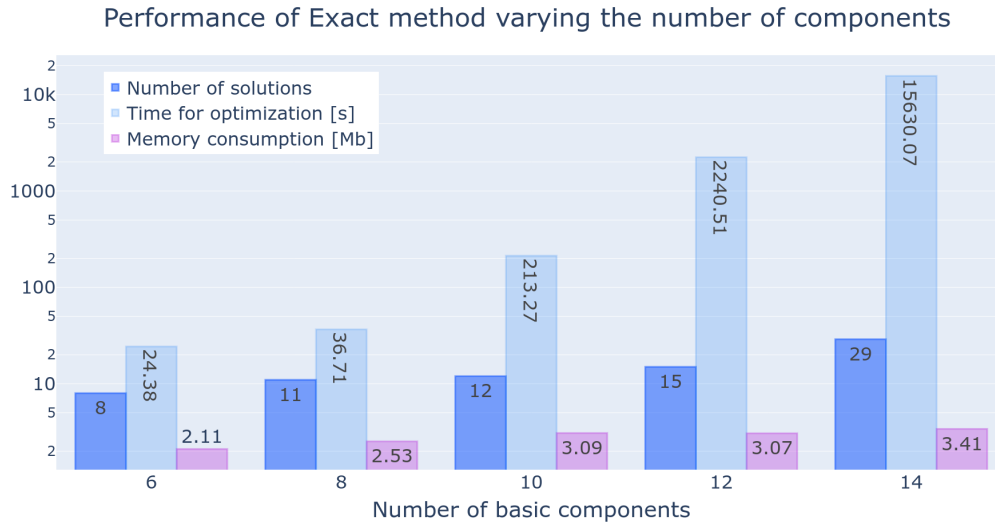


Figure 7.16: Time and memory performance of exact method varying the number of basic components.

7.5.1 Assessment Performance

The more burdensome parameter is reliability, as on the basis of our choice the other parameters are cumulative, and their assessment translates therefore in a simple addition. The overall performance of reliability assessment takes into account three main tasks:

- Abstraction (AllSMT computation),
- BDD-based quantifier elimination,
- BDD traversing.

Please note that the first two tasks are two types of *quantifier elimination*. The first one is performed through AllSMT, and it is applied to each CSA in order to quantify out non-Boolean variables. The second one is performed through BDD-based techniques to the entire architecture, in order to eliminate Boolean variables that abstract the inputs and the outputs of each pattern.

Task 1: Abstraction

The first task is facilitated by applying the predicate abstraction described above in Subsections 5.6.1 and 6.3.10. Furthermore, the caching mechanism described in Subsection 5.6.4 considerably improves the time performance. Thus, its contribute is negligible. However, note that the time required to perform the quantifier elimination task highly depends on the type of redundant

Pattern	# Inputs	# Models	Time [s]	Memory [KB]
<i>TMR_V111</i>	1	239	0.09	13.2
	2	2011	0.56	121.5
	3	16307	5.23	1100
<i>TMR_V012</i>	1	1149	0.36	70.8
	2	22849	3.48	782.7
	3	186953	38	7700
<i>TMR_V123</i>	1	2621	0.75	168.4
	2	22849	7.42	1600
	3	186953	78	14500

Table 7.13: Performance of quantifier elimination of the CSA of some redundant patterns.

pattern and on the arity of the basic component. For instance, the quantifier elimination of the CSA referred to the pattern *TMR_V123* (i.e., a TMR with three voters) is much slower than the CSA of the pattern *TMR_V111* (i.e., a TMR with one voter) because of the higher number of SMT variables modeling it. In fact, the former has two more voters than the latter, and each voter is modeled by a fault variable and a set of input and output ports. The more complex a pattern is, the higher is the number of possible behaviours described by its CSA formula, and therefore the higher is the number of its models. Table 7.13 reports the number of models, time performance, and memory consumption retrieved by the quantifier elimination. It clearly shows that it depends on the number of modules composing the pattern, and on the arity of the basic components, which influences the number of variables used to model the connected concretizer.

Once the AllSMT procedure retrieves the list of all models (which are exponential compared to the number of used variables), such models are aggregated in a DNF formula that has a size proportional to the number of truth assignments satisfying the CSA formula. This formula encodes all the deviations of the system from the nominal behavior. It consists of Boolean variables used to abstract the Real input ports and output ports of each pattern (i.e., the Boolean inputs of each concretizer and the Boolean outputs of each Abstractor). Since the BDD representing the CSs of the possible alternatives designs has to be made up of only configuration variables and fault variables, all the others Boolean variables have to be quantified out. This task can be accomplished by employing a BDD-based quantifier elimination procedure that is

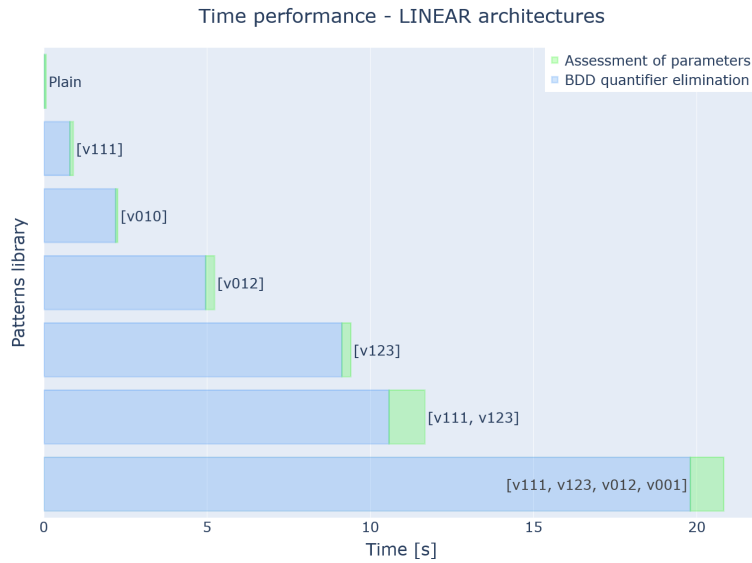


Figure 7.17: Time performance for extraction of non-functional parameters using a library of different redundant patterns for serial architecture of 100 components.

applied on the Boolean formula and abstracts the unnecessary Boolean atoms.

Task 2: BDD-based Quantifier Elimination

As expected, results indicated that the time for the BDD-based quantifier elimination grows with the number of components, the size of the pattern library, and on how components are connected, as illustrated in Figure 7.17 and in Figure 7.18, in which the labels on the vertical axis represent the different fan-out of the TMR patterns employed into the library. As a consequence, the performance is proportional to the BDD complexity.

Time performance on linear and rectangular architectures is linear with respect to the size of the structure, as illustrated in Figure 7.19, while using a library of instances of the same pattern type, the time is constant, as illustrated in Figure 7.20.

Conversely, for complex architectures, the curves become steeper already with few components (less than ten) if using more than five redundant patterns for each of them (see Figure 7.23).

Please note that in our experiments we used a static ordering of BDD variables, as described in Sub-section 6.3.10. In order to improve the performance related to this task, we can try and apply a dynamic reordering algorithm on the OBDD before the quantifier elimination. Table 7.14 shows how the size of the OBDD changes by varying the number of nodes in a linear and

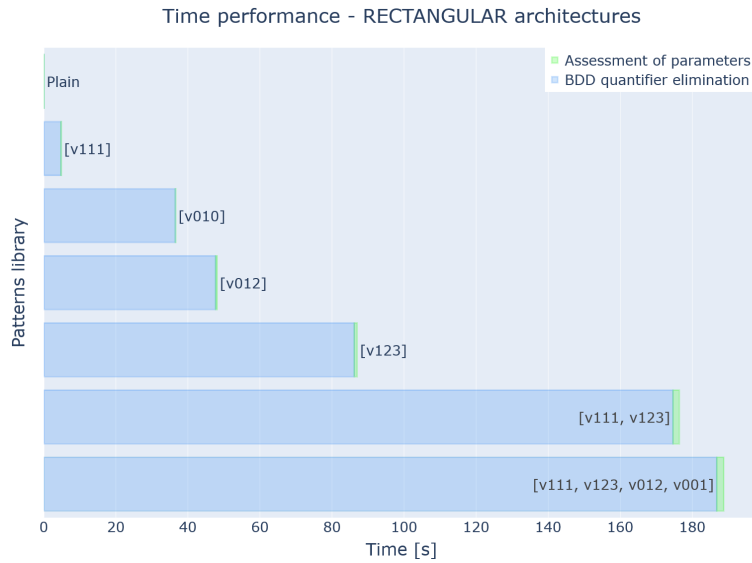


Figure 7.18: Time performance for extraction of non-functional parameters using a library of different redundant patterns for rectangular architecture of 100 components, organized in 50 levels.

rectangular architecture topology (the library of redundant patterns includes a TMR_V111 and a TMR_V123 for each component).

Figure 7.21 indicates that as the size of the system increases, the time needed to dynamically reduce the BDD grows much faster than the time required for the quantifier elimination. Furthermore, in this phase the OBDD is made up also of input and output Boolean variables. Hence, the time required for the reordering algorithm would increase even more. To sum up, although dynamic reordering significantly reduces the size of the OBDD, it requires long execution times, especially for very large and complex systems.

Topology	Ordering	1 level	2 levels	3 levels	4 levels	5 levels
Linear	Static (Ordering 1)	15	53	129	281	585
	Static (Ordering 3)	15	40	65	90	115
	Dynamic: SIFT	11	32	49	66	83
	Dynamic: Partial SIFT	11	30	47	64	81
Rectangular	Static (Ordering 1)	41	407	2015	8447	34175
	Static (Ordering 3)	28	129	266	403	540
	Dynamic: SIFT	21	98	210	309	408
	Dynamic: Partial SIFT	19	210	211	315	419

Table 7.14: Number of nodes of the OBDD by varying the ordering strategy. Ordering 1: all configuration variables on top of the OBDD, Ordering 3: driven by architecture.

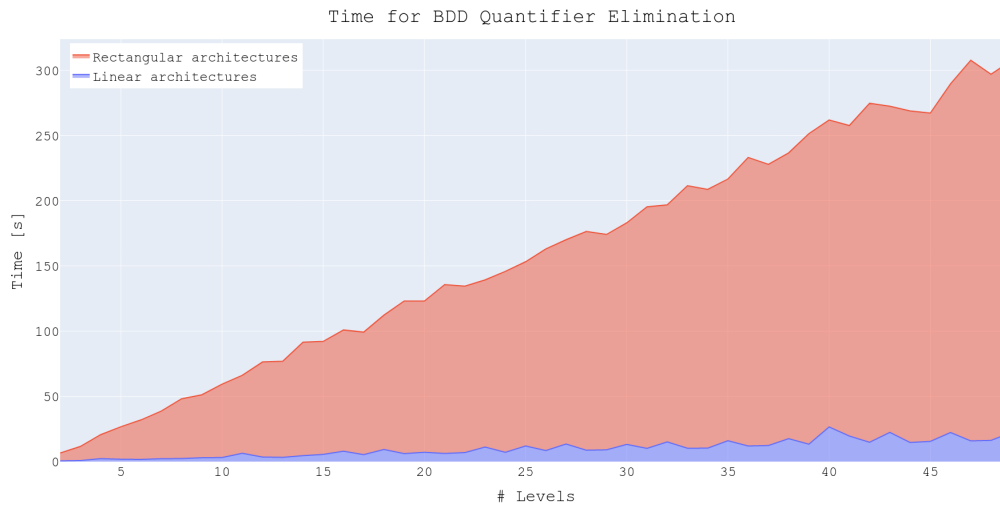


Figure 7.19: Time performance of BDD-based quantifier elimination using different redundant patterns, for linear and rectangular architectures.

Task 3: BDD Traversing

The performance of the reliability function extraction is linear with respect to the size of the BDD, as illustrated in Figure 7.22. Conversely, for randomly generated (i.e., complex) systems the curves grow faster because the topology no longer has a periodicity, as illustrated in Figure 7.23. As expected, the time curves grow faster as we add more components to the system and/or

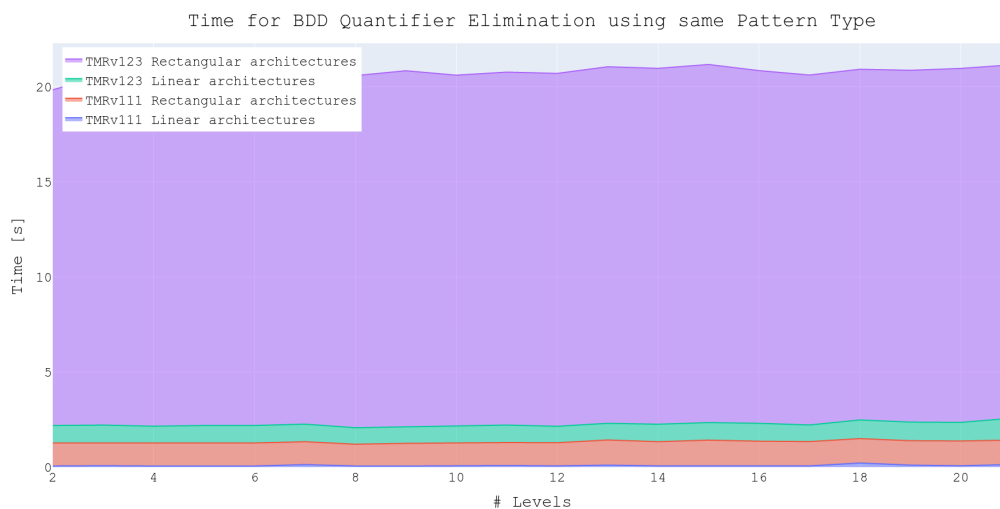


Figure 7.20: Time performance of BDD-based quantifier elimination using different instances of the same redundant patterns, for linear and rectangular architectures.

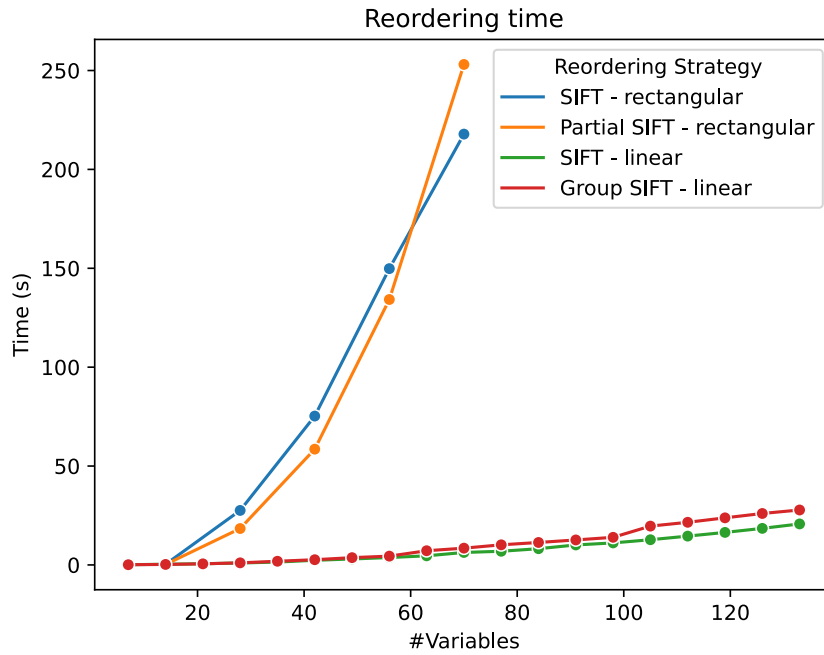


Figure 7.21: Time performance of different types of dynamic reordering strategies for linear and rectangular architectures.

more redundant patterns to the libraries. Anyway, once the OBDD has been created, the reliability extraction is very efficient even with the most complex systems.

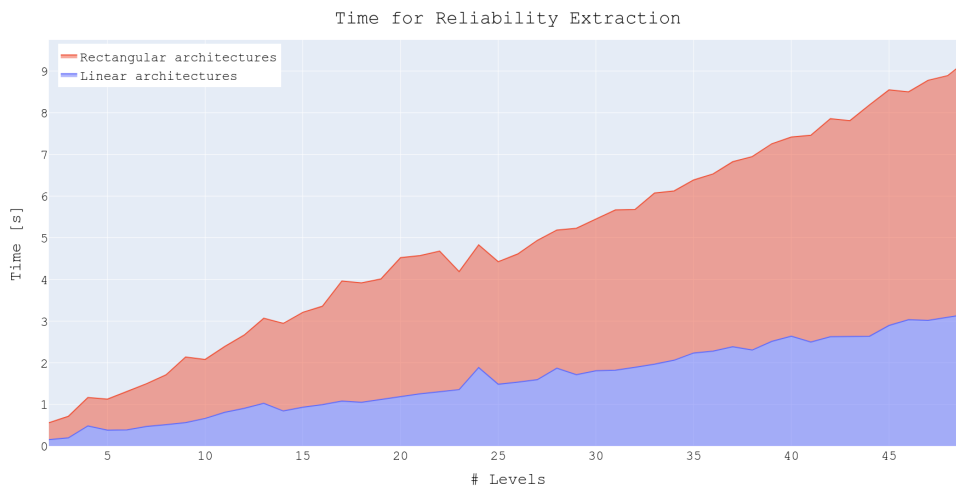


Figure 7.22: Time performance of reliability function extraction for linear and rectangular architectures, using different types of patterns.

Figure 7.24 shows the total time needed to extract the reliability formula in random topologies with different number of components. For each size, ten different random architectures have been generated, and for each of them the reliability has been extracted.

Figure 7.25 shows the performance for complex systems. We also investigated different BDD variable orderings, using the SIFT algorithm. Unfortunately, with complex architectures, the application of a reordering strategy adds a considerable overhead, resulting inapplicable for architectures consisting of dozens of components. For this reason, we preferred a static ordering according to the topology of the basic architecture.

7.5.2 Optimization Performance

As far as the optimization is concerned, the results indicated that with the enumerative approach the time needed for reliability extraction grows very fast when the size of the architecture increases. This happens because the number of alternative architectures is combinatorial with respect to the number of components, and we need to extract the reliability separately for each one of them.

The symbolic approach allows an efficient encoding of the reliability, however the complexity of this representation significantly slows down the time needed for the optimization task.

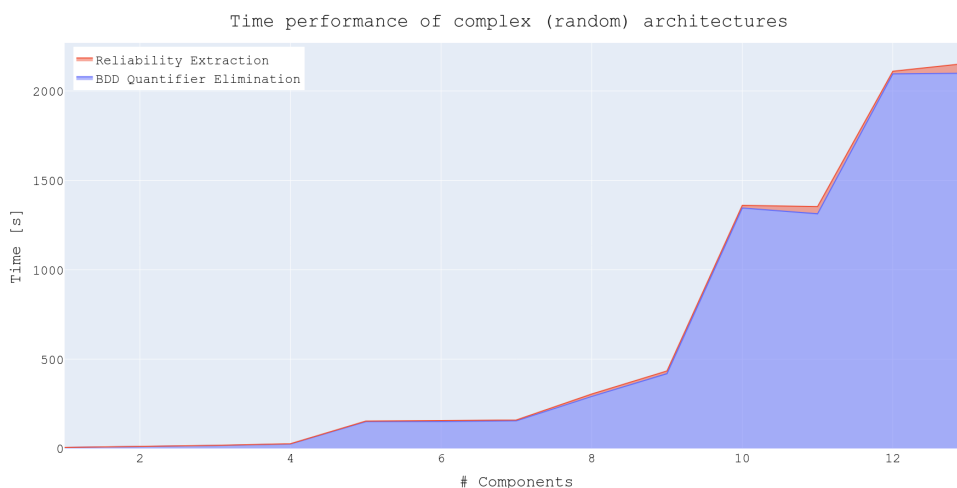


Figure 7.23: Time performance for BDD-based quantifier elimination and reliability function extraction for complex architectures.

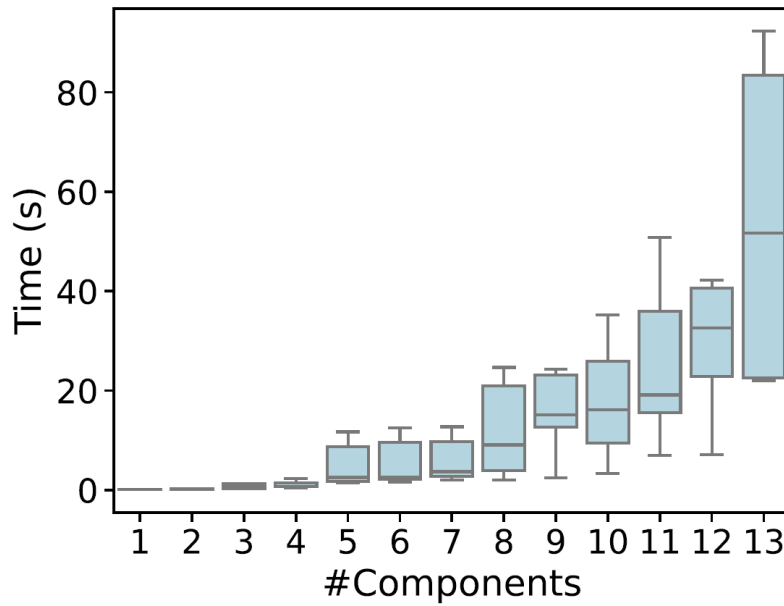


Figure 7.24: Time performance for reliability function extraction of complex architectures (d is the maximum incoming degree of the components, i.e., the maximum number of incoming connections).

The hybrid approach outperforms the symbolic one, as reported in Table 7.15, because it allows us on the one hand to leverage the power of the symbolic extraction of the reliability, and on the other hand to efficiently extract the design points by feeding the optimizer with an explicit function.

In addition, the results suggested that time performance is influenced mainly by the number of components and patterns, rather than by the number of objective functions. Indeed, adding more objectives to the problem produced a not relevant overhead (see Figure 7.26). This is mainly due to the fact that except for the reliability, all cost functions were considered as cumulative.

Table 7.15: Optimization: symbolic vs hybrid approach.

Topology	Linear		Rectangular	
	Symbolic	Hybrid	Symbolic	Hybrid
Length 1	0.05s	0.04 s	5.85 s	0.07 s
Length 2	4.09 s	0.15 s	645 s	0.43 s
Length 3	153 s	0.32 s	>5000 s	2.67 s

Time performance - COMPLEX (RANDOM) architectures

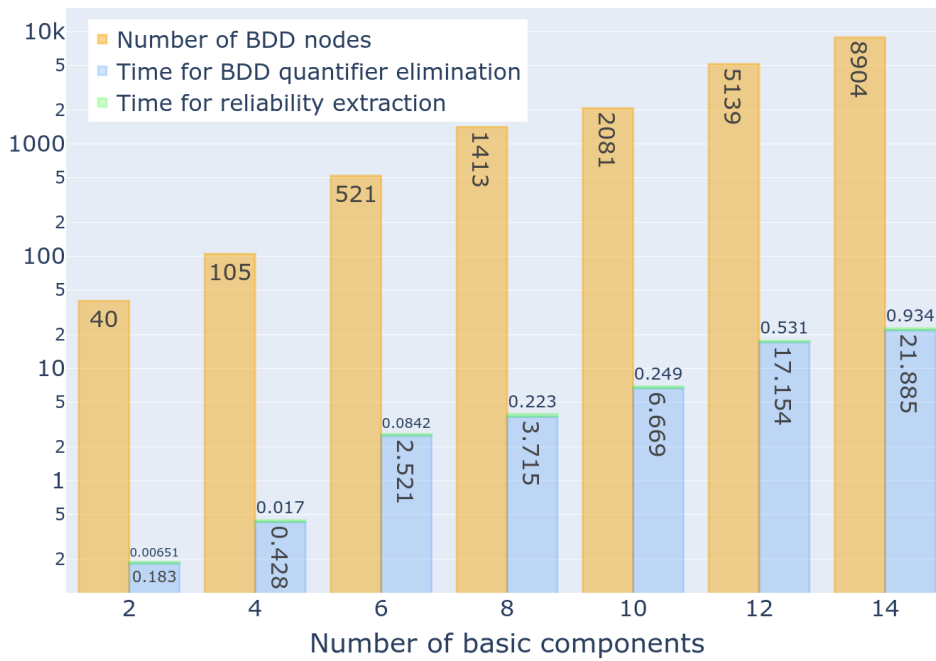


Figure 7.25: Number of BDD nodes, time performance of BDD-based quantifier elimination, and reliability function extraction for complex architectures varying the number of components.

Time performance varying the number of objectives

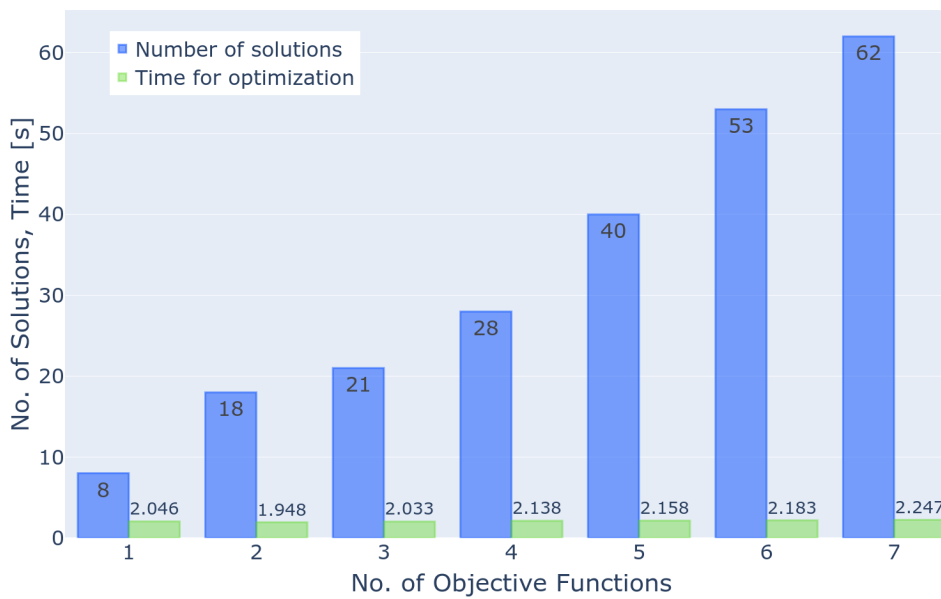


Figure 7.26: Performance of optimization when varying the number of objectives (system of 6 components with 3 patterns each).

7.6 Test problem

In order to compare the performance of our method for reliability assessment with other existing methods, we faced a test problem introduced by Beccuti et al. [308]. The benchmark is a system with two hundred twenty-three nodes and two hundred fifty-two edges connected as illustrated in Figure 7.27. In their work, the authors derive the set of min-paths/min-cuts from the network and encode them on a BDD. Then, they derive another BDD encoding the connectivity function. Finally they compute the reliability, defined as the probability that the source node is connected to the terminal node by at least one path of working edges. Hence, they assume that only edges can fail (while we reason on nodes). In addition they assume that all edges have same probability, and all components are identical. They compute the exact reliability value by decomposing the system into as many sub-systems as the number of repeating patterns with one source and one destination, solving each sub-system in isolation, and then computing the reliability as the product of their reliabilities (since they are connected in series). They also compute an approximate value of the system reliability by considering only a subset of all the min-paths/min-cuts. The user can specify constraints on execution times, so that only the min-paths/min-cuts generated under these constraints are inserted in the BDDs. Table 7.16 reports the results of their heuristic.

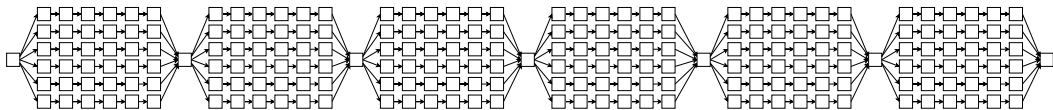


Figure 7.27: Topology used in the Test Problem, with 223 nodes and 252 edges.

Table 7.16: Results of the method proposed by Beccuti et al. [308]

Min-paths	Constraints	Execution Time	Min-cuts	Constraints	Execution Time
46656	4 s - 45 m	32 m	19223	30 s - 6 m	37 s

By setting the same fault probability for all nodes, and considering a pattern library composed of plain components only (i.e., we just want to compute the reliability of the given architecture, with no redundancy), our method found the exact solution for the entire system with the following performance:

Time elapsed: 0.5719640550000236 s

Memory usage: 2.3109474182128906 Mb.

Considering only one stage, in the same manner of the cited work, memory usage and time were further improved:

Time elapsed: 0.11293733299999076 s
Memory usage: 2.154712677001953 Mb.

Memory usage is the same, but time performance was reduced.

7.7 Applicability and Limitations

The main advantage of this method is the encoding of the reliability search problem to symbolically explore the performance of different redundant schemes. The approach automatically extracts a symbolic reliability function that maps the probability of fault of the basic components to the probability that the overall architecture deviates from the expected behavior. The benefit of the symbolic encoding of the architecture-based reliability evaluation is evident, but it could add complexity to the models.

The main challenge arises when the number of system components and patterns is too high, and their connections is very complex, due to the sheer size of the design space. Enumerating every point is prohibitive. On large optimization problems, the computational time increases strongly with the instance size. In addition, the excessive memory consumption can lead to early abortion. We encounter this issue with complex architectures made up of dozens of nodes and pattern libraries with more than six patterns for each basic component, as showed in Section 7.4.5. For the above reason, in the following we propose a meta-heuristic optimization approach for exploring the design space in a cost-effective manner, reducing the complexity and fostering parallelization.

Chapter 8

Near-Optimal Approximations

Despite known successes, exact methods have also some disadvantages. First of all, the computational time increases strongly with the instance size. In addition, the memory consumption can be very large and lead to early abortion. In this chapter, we propose a slight modification to the exact algorithm proposed above in order to achieve good performance for solving our RAP.

8.1 Simplifying the Exact Method

On large optimization problems, approximate search algorithms find a solution more quickly, although their weakness is that they are not able to determine whether a solution exists or not, and they have no guarantee to find a solution if it exists. Anyway, our case is of practical application and input formulae are actually expected to be satisfiable, making them well-suited for incomplete search algorithms.

The first idea that came to mind was to integrate both techniques by performing a sequential execution of an exact method applied to (simpler) sub-problems that is launched before a meta-heuristic, in order to balance between local and global search. For instance, instead of considering a single problem with four different design objectives, we could consider two sub-problems with

two objectives each, using information coming from solutions of relaxations of initial formulation to provide good initial solutions for a local search, and restrict the area to be examined in the second phase. The meta-heuristics would work on a problem that has a different nature from the considered optimization problem. The information obtained from the relaxation should help restrict the search space to areas where optimal solutions are located. However, since the main bottleneck of our method is the number of components and patterns rather than the number of objectives, we propose an alternative method for complexity reduction and parallelization in the following. It is based on graph partitioning, and the underlying idea is to partition the original system into two or more parts, apply the exact method presented above to each part, and then combine back the solution for the original system.

8.2 Graph Partitioning

A common method to face complex problems leverages graph partitioning, i.e., the reduction of the graph representing the system under study into smaller graphs, by partitioning its set of nodes into mutually exclusive groups. This issue can be posed as the following *Graph Partitioning Problem* (GPP). Given a graph $G = (V; E)$, where nodes represent system's components and edges represent data communication, the goal is to divide V into equal sized parts V_1, \dots, V_k while minimizing the edges cut. In other words, we want to split the original system into two or more sub-systems while minimizing dependencies from the point of view of reliability. Using this approach, first of all we can simplify the reliability extraction, in addition we can parallelize the computation of the cost functions. Several variants of this problem exists, and several solutions have been proposed, which range from very fast heuristics or simple algorithms based on *Breadth First Search* (BFS) to sophisticated combinatorial optimization methods. We are going to use the method of partitioning with special attention to reliability, which is highly dependent on how components are connected, and consequently, on how the original system is partitioned. To this aim, we employ two well-known algorithms, presented in the following.

8.2.1 Kernighan–Lin Algorithm

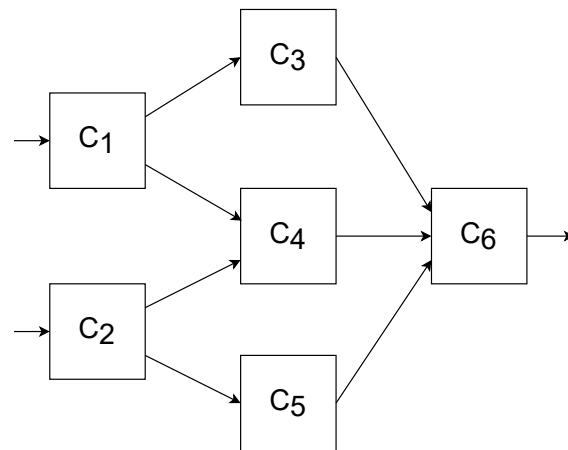
The KL algorithm is a heuristic algorithm for finding partitions of graphs. The input to the algorithm is an undirected graph $G = (V; E)$ with vertex set V ,

edge set E , and (optionally) numerical weights on the edges. The goal of the algorithm is to partition V into two disjoint subsets A and B of (nearly) equal size, in a way that minimizes the sum T of the weights of the subset of edges that cross from A to B . If the graph is unweighted, then instead the goal is to minimize the number of crossing edges (this is equivalent to assigning weight one to each edge).

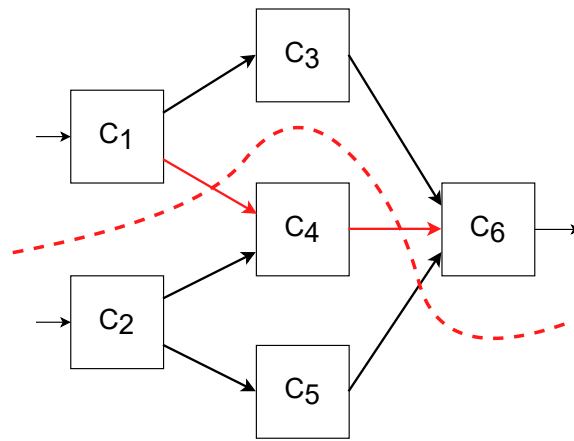
Let us consider our running example made of six components. Figure 8.1a shows the DAG with six vertices and seven edges corresponding to computing dependencies. An edge $(C_i; C_j)$ implies that C_j depends on C_i . If the edge orientations are removed and the computation is modeled with an undirected graph, the resulting balanced partition with a 3/3 vertex split could have the edges in the cut producing a *cyclic inter-dependency* between the two parts, as illustrated in Figure 8.1b. Inter-dependency means that the failure of an element in one partition may cascade to the other partition and cause the failure of dependent elements. In order to minimize the dependencies for reliability calculation, we are interested in acyclic partitions, like the one illustrated in Figure 8.1c. The KL algorithm is based on partition via exchange, i.e., exchange the set of vertices of a given graph between the two parts, while minimizing the cut-edges. It repeats this task until there are no exchanges that optimize the cut-edges function.

8.2.2 Multi-level Partitioning

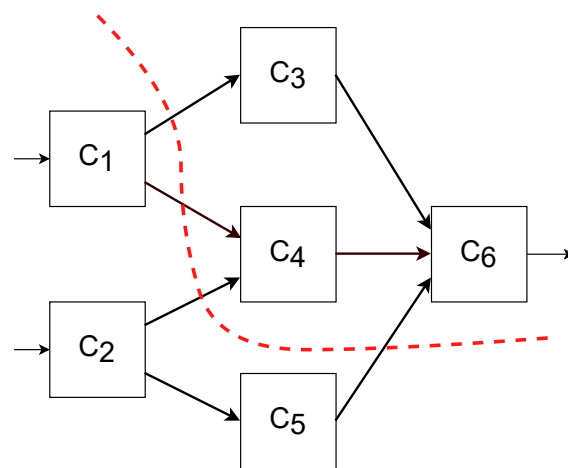
One of the most successful heuristics for partitioning large graphs is the *Multi-level Graph Partitioning* (MGP) approach. It consists of the three main phases: coarsening, initial partitioning, and uncoarsening. The main goal of the coarsening (aka contraction) phase is to gradually approximate the original problem and the input graph with fewer degrees of freedom. Coarsening is usually stopped when the graph is sufficiently small to be initially partitioned using some algorithm. Uncoarsening consists of two stages. First, the solution obtained on the coarse level graph is mapped to the fine level graph. Then, the partition is improved by using some heuristics (e.g., local search) that iteratively improve the solutions. The multi-level approach works so well for at least three intuitive reasons. Firstly, at the coarse levels a lot of work per node can be performed without increasing excessively the overall execution time. Furthermore, a single node move at a coarse level corresponds to a big change in the final solution. Hence, we might be able to find improvements easily that



(a)



(b)



(c)

Figure 8.1: (a) A toy example with 6 vertices and seven edges, (b) cyclic 2-way partitioning (c) acyclic partitioning of the same directed graph.

would be difficult to find on the finest level. Finally, fine level local improvements are expected to run fast since they already start from a good solution inherited from the coarse level. We used the tool named Metis proposed by Karypis and Kumar [309], which implements the above concepts.

8.3 Partitioning the System Architecture

The approximate method is illustrated in Figure 8.2. Compared to the exact method, it introduces some new steps. First of all, the graph representing the given architecture is partitioned in two or more parts depending on its size. Experimentally, we found that with complex architectures when the basic system has more than fourteen components and more than six redundant patterns for each component, computing time increases excessively (more details follow in the experimental evaluation). For this reason, we use the following heuristic as partition rule: if the basic system is composed of a number of components less than fifteen we use KL algorithm, if the basic system is composed of a number of components greater than or equal to fifteen we use Metis algorithm. Afterwards, we can apply the exact method as presented in Chapter 6 for each sub-architecture obtained from partitioning. The result is a set of sub-solutions for each partition of the given system.

8.4 Combining Solutions from Sub-architectures

In order to create and evaluate the solutions for the basic system architecture, the sub-solutions are combined together. We combine them by building a formula made up of a conjunction of the relative configuration variables, in order to obtain a set of solutions for the entire architecture. This formula and the cost formulae of the entire system are then provided as assertions to a solver that retrieves the models for the original architecture.

8.5 Pruning and Ranking for Large Problems

The approximate solutions are the result of a flattening process that produces all possible combinations of the exact solutions of the various partitions. If we have several partitions, the set of approximate solutions will contain every ordered combinations of all the partitions elements. Since the number of this

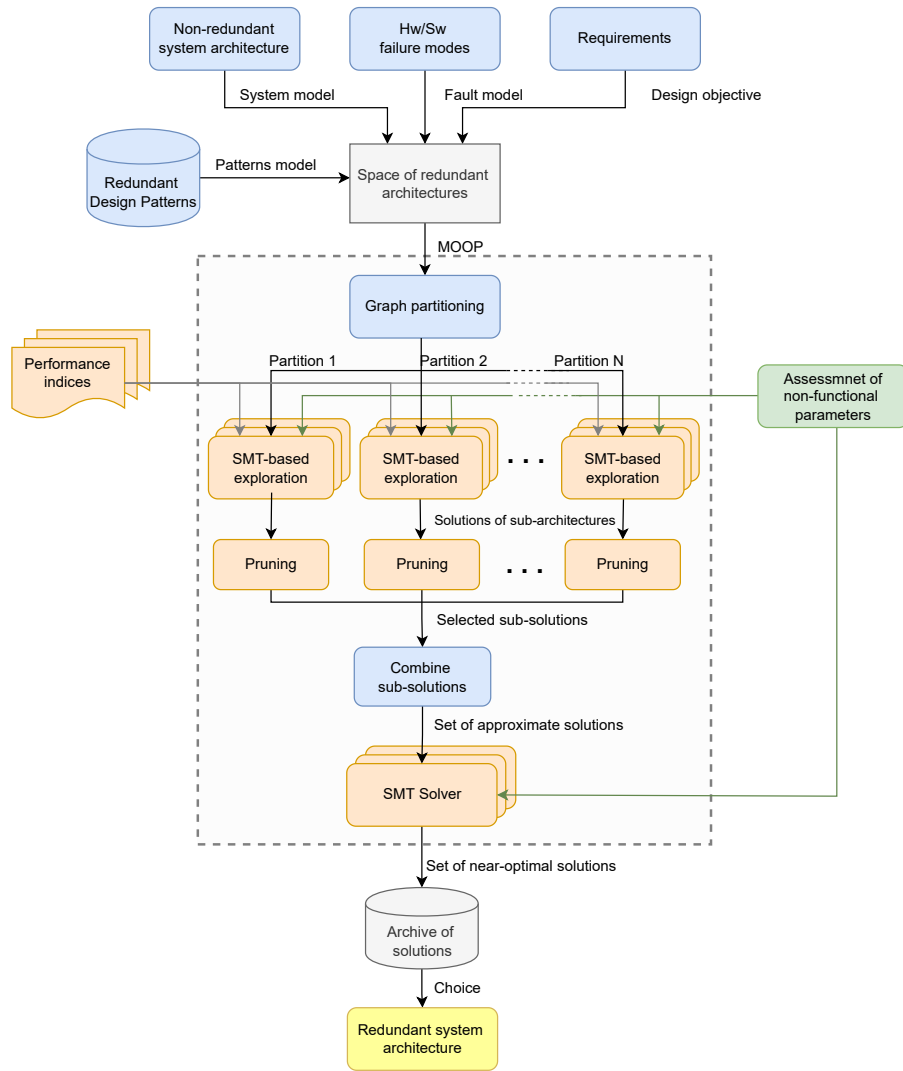


Figure 8.2: Approximate method for DSE of redundant architectures, overall process flow.

set can increase rapidly, a pruning technique is advised, in order to reduce the size of the search area by removing less promising sections. The pruning and ranking methods basically select the most relevant Pareto solutions. The literature is extensive as there are numerous methods that can for example be classified on the type of data they use. Here, we introduce a simple pruning rule, before combining the sub-solutions. For each partition, we discard the solutions with each cost over a certain threshold th . This threshold is variable and can be found empirically starting from the average solution avg_{cost} and gradually reducing or increasing the distance Δ from it, as follows:

$$th = avg_{cost} \pm \Delta$$

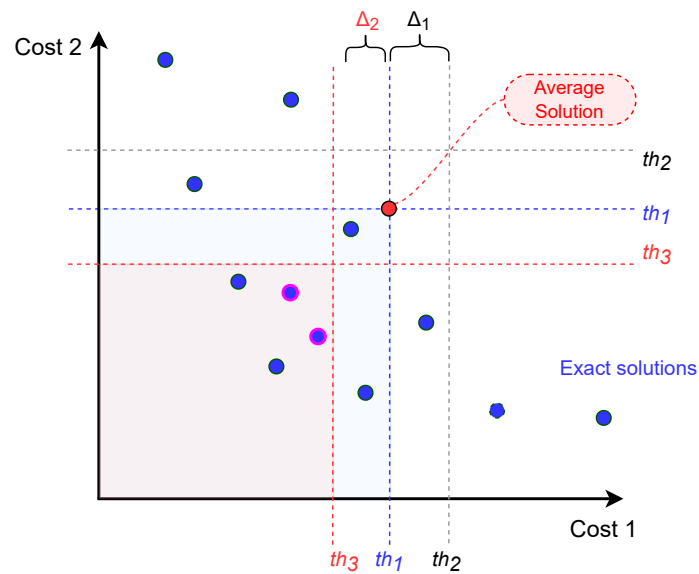


Figure 8.3: The pruning strategy allows us to adapt the search area by varying a threshold for solution acceptance: all the solutions above the threshold are discarded. This is a flexible solution that can be tuned on the basis of problem instance size and resources available.

thus, reducing or enlarging the search area, as illustrated in Figure 8.3 for a problem with two objective functions. In this example, setting the threshold to th_2 allowed us to identify eight solutions out of twelve total solutions, while setting it to th_3 reduced the searching area, missing some solutions (we obtained four solutions), but speeding up the process. In addition, discarding dominated solutions (those circled in violet) reduced the final set to two solutions. The average costs can also be asymmetric, in order to favor extremal or medial variables with reference to the Pareto front, if wanted. Afterwards, we scan iteratively all solutions, discarding the dominated ones. The advantage of this method is that it is very flexible: we can adapt the search area to the instance size of the problem and to resources available, considering more solutions if possible or speeding up the process with strict thresholds in case of very large systems. Please note that when using pruning there is a risk of missing some “good” global solutions. To address this issue, we favoured symmetric cost thresholds that would guarantee balanced solutions, and used the smallest number possible of partitions that would lead to reasonable computing times, in order to limit the explosion of the number of global solutions. Further details are provided in Section 9.5. The final algorithm is outlined in Algorithm 3.

Algorithm 3 DSE general framework

```

1: Input 1: system architecture model
2: Input 2: library of redundant patterns
3: Input 3: fault model
4: Input 4: design objectives
5: [OPT] Input 5: additional constraints (with local or global scope)
6: Output: set of optimum redundant architectures
7:
8: Phase 1 - Modeling
9: Define configuration variables  $cfg_i = (C_i, P_j)$  and fault variables  $F_i$ 
10: Define the behaviors of the components through SMT constraints
    a: For each CSA use basic version as nominal behavior
    b: For each  $C_i$ , find the  $P_j$  with highest number  $F_i$  for applying variable
        sharing
11: Define the linking constraints  $\triangleright$  SMT formula of the redundant
    architectures
12: Define configuration constraints  $\text{len}(cfg_i) = \lceil \log_2(\text{len}(lib_{C_i})) \rceil$ 
13: Define the compatibility constraints through SMT constraints
14: Phase 2 - Graph partitioning
15: if  $\text{number}(C_i) > 14$  then
16:      $method = METIS$ 
17: else  $method = KERNIGHAN - LIN$ 
18: end if
19: Compute partitions
20: for each partition in partitions do
21:     Phase 3 - Assessment
22:     Miter composition
23:     Computation of MCSs  $\triangleright$  AllSMT computation using MathSAT solver
24:     Caching the resulting formula
25:     Conversion of the formula into a BDD representation.
26:     BDD-based quantifier elimination.
27:     Symbolic Reliability function extraction  $\triangleright$  by BDD traversing
28:     Assessment of other non-functional parameters
29:     Phase 4 - Optimization
30:     if  $approach = ENUMERATIVE$  then
31:         Compose a new Miter for each alternative
32:         Compute the MCs for each alternative
33:         Create a mapping configuration  $\rightarrow$  cost functions
34:     else if  $approach = SYMBOLIC$  then
35:         Parameterize the system
36:         Compute the values at run-time  $\triangleright$  using Z3 solver
37:     else  $approach = HYBRID$ 
38:         Perform semi-symbolic choices
39:     end if
40:     Pruning and selection of solutions
41: end for
42: Phase 5 - Combine sub-solutions
43: Combining solutions of sub-architectures
44: Evaluate combined solutions for the global system  $\triangleright$  using Z3 solver

```

8.6 Running Example

To help illustrate the above concepts, we applied the approximate method proposed to the running example. We considered three different partitions, as illustrated in Figure 8.4. In this case, since the system is composed of only six components, we used the KL algorithm for partitioning (obtaining only two partitions).

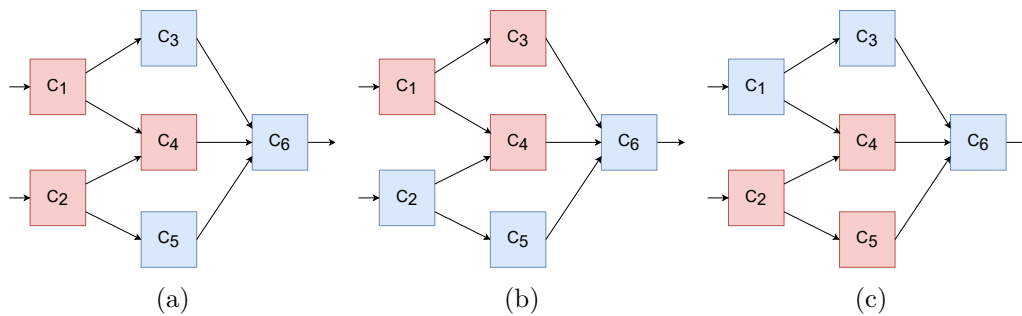


Figure 8.4: Three different partitions for the running example, using KL algorithm.

The approximate method using the partitioning of Figure 8.4a found eighteen solutions reported in Table 8.1 (including the eight optimal ones retrieved with the exact method), produced by flattening the two sets of solutions reported in Table 8.2.

Applying the pruning strategy to the the two partitions of Figure 8.4a, we obtained respectively four and one solutions, as reported in Table 8.3. By flattening these solutions we obtained a total of four solutions for the original architecture, reported in Table 8.4. Figures 8.5a and 8.5b illustrate how the pruning strategy selects more promising solutions. Table 8.5 reports the comparison of solutions.

Figure 8.6 and Figure 8.7 show the comparison of exact and approximate solutions without and with pruning strategy for the three partitionings considered.

Partitioning of Figure 8.4c generated cheaper solutions, while that of of Figure 8.4a generated solutions with higher reliability.

The results presented in Chapter 9 will indicate that for a system with few components the exact method is preferable to the approximate one as this takes more time since it needs two calls to the solver for the optimization of the two partitions, and one more to evaluate the models of the original architecture.

Table 8.1: Approximate solutions for system in Figure 8.4a (considering two objective functions).

Solution	Fault probability	Cost
1	0.6739644436312434	225.0
2	0.6929706872988478	222.0
3	0.7024293686419650	224.0
4	0.7058063295659203	221.0
5	0.7126954744932213	221.0
6	0.7229563504041777	218.0
7	0.7369794102422135	220.0
8	0.7460535490548722	217.0
9	0.7533483595885260	218.0
10	0.7677269176540538	215.0
11	0.7744199305735989	217.0
12	0.7822023816175623	214.0
13	0.7912489351095497	221.0
14	0.8086525638205954	220.0
15	0.8154864008270026	217.0
16	0.8356651111717405	216.0
17	0.8417515885357502	214.0
18	0.8545405196347040	213.0

Table 8.2: Solutions of the two partitions of system in Figure 8.4a

Partition 1			
Solution	Fault probability	Cost	Redundant patterns
1	0.4797150862035774	117	$C_3 : Tmr_V123, C_5 : Tmr_V123, C_6 : Tmr_V123$
2	0.5230913701445632	116	$C_3 : Tmr_V123, C_5 : Tmr_V111, C_6 : Tmr_V123$
3	0.5401238212108288	113	$C_3 : Tmr_V123, C_5 : Tmr_V123, C_6 : Tmr_V111$
4	0.59041663565824	112	$C_3 : Tmr_V123, C_5 : Tmr_V111, C_6 : Tmr_V111$
5	0.60558638989312	110	$C_3 : Tmr_V111, C_5 : Tmr_V123, C_6 : Tmr_V111$
6	0.637461139456	109	$C_3 : Tmr_V111, C_5 : Tmr_V111, C_6 : Tmr_V111$
Partition 2			
Solution	Fault probability	Cost	Redundant patterns
1	0.43558377655120223	108	$C_1 : Tmr_V123, C_2 : Tmr_V123, C_4 : Tmr_V123$
2	0.48263929886218243	105	$C_1 : Tmr_V123, C_2 : Tmr_V111, C_4 : Tmr_V123$
3	0.58310264700928	104	$C_1 : Tmr_V111, C_2 : Tmr_V111, C_4 : Tmr_V123$

But when the number of components begins to grow, the approximate method outperforms the exact one.

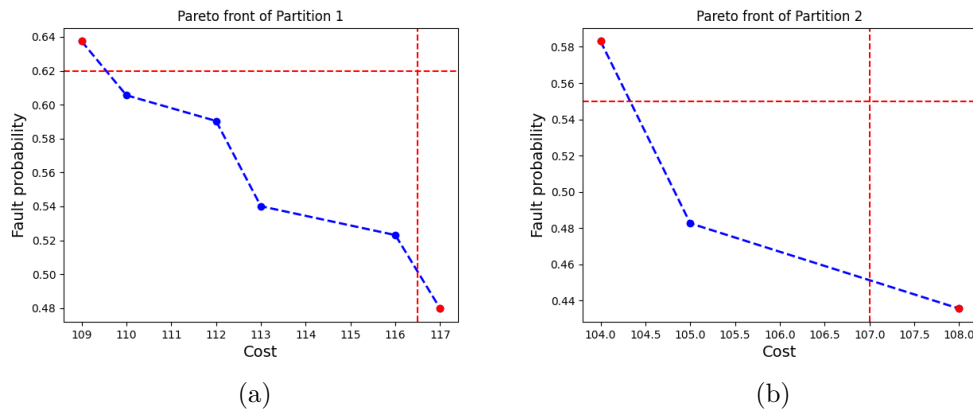


Figure 8.5: Applying the pruning strategy to the two partitions of our running example lead to the selection of four solutions for the first partition and one solution for the second one.

Table 8.3: Solutions of the two partitions of system in Figure 8.4a after pruning

Partition 1			
Solution	Fault probability	Cost	Redundant patterns
1	0.5230913701445632	116	$C_3 : Tmr_V123, C_5 : Tmr_V111, C_6 : Tmr_V123$
2	0.5401238212108288	113	$C_3 : Tmr_V123, C_5 : Tmr_V123, C_6 : Tmr_V111$
3	0.59041663565824	112	$C_3 : Tmr_V123, C_5 : Tmr_V111, C_6 : Tmr_V111$
4	0.60558638989312	110	$C_3 : Tmr_V111, C_5 : Tmr_V123, C_6 : Tmr_V111$
Partition 2			
Solution	Fault probability	Cost	Redundant patterns
1	0.48263929886218243	105	$C_1 : Tmr_V123, C_2 : Tmr_V111, C_4 : Tmr_V123$

Table 8.4: Approximate solutions for example system in Figure 8.4a

Solution	Fault probability	Cost
1	0.7126954744932213	221.0
2	0.7229563504041777	218.0
3	0.7460535490548722	217.0
4	0.7677269176540538	215.0

Table 8.5: Comparison of solutions for system in Figure 8.4a (considering two objective functions)

Solution	Exact solutions		Approximate solutions		Approximate solutions with Pruning	
	Fault probability	Cost	Fault probability	Cost	Fault probability	Cost
1	0.6739644436312434	225	0.6739644436312434	225	-	-
2	0.6929706872988478	222	0.6929706872988478	222	-	-
3	-	-	0.7024293686419650	224	-	-
4	0.7058063295659203	221	0.7058063295659203	221	-	-
5	-	-	0.7126954744932213	221	0.7126954744932213	221
6	0.7229563504041777	218	0.7229563504041777	218	0.7229563504041777	218
7	-	-	0.7369794102422135	220	-	-
8	0.7460535490548722	217	0.7460535490548722	217	0.7460535490548722	217
9	-	-	0.7533483595885260	218	-	-
10	0.7677269176540538	215	0.7677269176540538	215	0.7677269176540538	215
11	-	-	0.7744199305735989	217	-	-
12	0.7822023816175623	214	0.7822023816175623	214	-	-
13	-	-	0.7912489351095497	221	-	-
14	-	-	0.8086525638205954	220	-	-
15	-	-	0.8154864008270026	217	-	-
16	-	-	0.835665111717405	216	-	-
17	-	-	0.8417515885357502	214	-	-
18	0.8545405196347040	213	0.8545405196347040	213	-	-

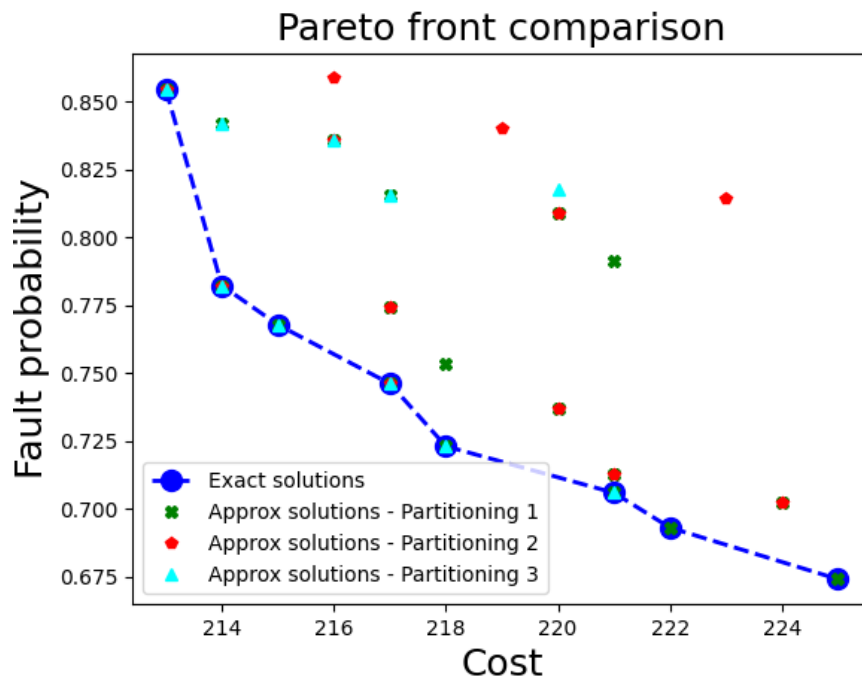


Figure 8.6: Comparison of approximate solutions for the three partitionings in Figure 8.4 without pruning.

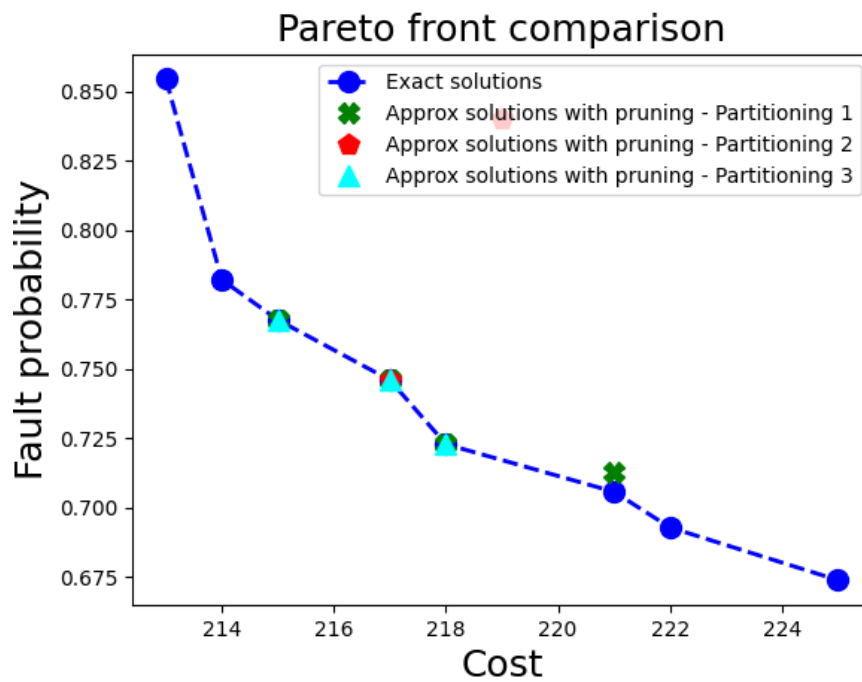


Figure 8.7: Comparison of approximate solutions for the three partitionings in Figure 8.4 with pruning.

Chapter 9

Experimental Evaluation of Approximate Method

In this chapter we validate the approximate method, highlighting how it can face problems that are prohibitive for the exact method illustrated in Chapter 6. In addition we compare it with existing related works.

9.1 Implementation details

The KL heuristic algorithm employed in our method is the one included in the NetworkX package (see Figure 9.1). It partitions a given architecture into two sets by iteratively swapping pairs of nodes to reduce the edge-cut between the two sets. The pairs are chosen according to a modified form of the original KL algorithm, which moves node individually, alternating between sides to keep the bisection balanced. The source code is reported in Figure 9.2.

For MGP we employed the already mentioned METIS package that implements various multi-level algorithms. We use it through the Python wrapper named PyMetis [310]. It only wraps the most basic graph partitioning functionalities, but it is enough for our use.

kernighan_lin_bisection(*G*, *partition=None*, *max_iter=10*, *weight='weight'*, *seed=None*)

Partition a graph into two blocks using the Kernighan–Lin algorithm.

Parameters: **G** : *graph*

partition : *tuple*

Pair of iterables containing an initial partition. If not specified, a random balanced partition is used.

max_iter : *int*

Maximum number of times to attempt swaps to find an improvement before giving up.

weight : *key*

Edge data key to use as weight. If None, the weights are all set to one.

seed : *integer, random_state, or None (default)*

Indicator of random number generation state.

Returns: **partition** : *tuple*

A pair of sets of nodes representing the bipartition.

Raises: **NetworkXError**

If partition is not a valid partition of the nodes of the graph.

Figure 9.1: The KL algorithm included in the NetworkX package.

9.2 Benchmarks

In this Section, we present some of the experiments used for validating our method. Firstly, we show the application of the approximate method to the same experiments used for validating the exact one. In addition, we present some examples on larger systems.

9.2.1 Experimental setup

Compared to the exact method, the setting for the experimental evaluation comprises some additional tasks. First of all, the original architecture is partitioned into smaller sub-architectures. To each sub-architecture the exact method and subsequent pruning action are applied in sequence. Then, the solutions retrieved for each sub-architecture are combined together. Eventually, the solver is invoked again in order to retrieve the models for the original architecture.

9.2.2 Evaluation criteria

Since S-P systems can be easily decomposed in combinations of smaller S-P systems, and their reliability can be easily computed as seen in Section 2.2,

```

def kernighan_lin_bisection(G, partition=None, max_iter=10, weight="weight", seed=None):
    n = len(G)
    labels = list(G)
    seed.shuffle(labels)
    index = {v: i for i, v in enumerate(labels)}

    if partition is None:
        side = [0] * (n // 2) + [1] * ((n + 1) // 2)
    else:
        try:
            A, B = partition
        except (TypeError, ValueError) as err:
            raise nx.NetworkXError("partition must be two sets") from err
        if not is_partition(G, (A, B)):
            raise nx.NetworkXError("partition invalid")
        side = [0] * n
        for a in A:
            side[index[a]] = 1

    if G.is_multigraph():
        edges = [
            [
                (index[u], sum(e.get(weight, 1) for e in d.values()))
                for u, d in G[v].items()
            ]
            for v in labels
        ]
    else:
        edges = [
            [(index[u], e.get(weight, 1)) for u, e in G[v].items()] for v in labels
        ]

    for i in range(max_iter):
        costs = list(_kernighan_lin_sweep(edges, side))
        min_cost, min_i, _ = min(costs)
        if min_cost >= 0:
            break

        for u, v, (u, v) in costs[:min_i]:
            side[u] = 1
            side[v] = 0

    A = {u for u, s in zip(labels, side) if s == 0}
    B = {u for u, s in zip(labels, side) if s == 1}
    return A, B

```

Figure 9.2: Source code for the KL algorithm employed.

we are mainly interested in complex architectures. Furthermore, as concerns the optimization, we only use the hybrid approach as we have already showed that it outperforms the enumerative and symbolic ones.

The overall performance of the approximate method has four main contributions: the system partitioning, the application of the exact method to the individual partitions, the selection and pruning of sub-solutions, the assembly of the selected solutions for the original system and their evaluation. In order to get a rough measurement of the time complexity of the algorithm, the overall CPU time and memory usage are measured when running the experiments. Another evaluation parameter is the number of nodes of the BDDs generated, which can help figuring out the dimension and complexity of the system we are dealing with.

9.2.3 Experiments on complex architectures

In the following we report, among others, the outcomes of the same experiments we have used for the exact method, in order to perform a comparison between the two methods proposed. Afterwards, we provide example of larger architectures whose analysis is prohibitive with the exact method, but can be faced smoothly with the approximate one. Moreover, we evaluate the KL and Metis algorithms for partitioning, and illustrates the effects of the pruning strategy employed.

Example system with 8 components

Recall the example system composed of eight components that we have introduced in Section 7.4.5, and consider the KL partitioning of Figure 9.3a.

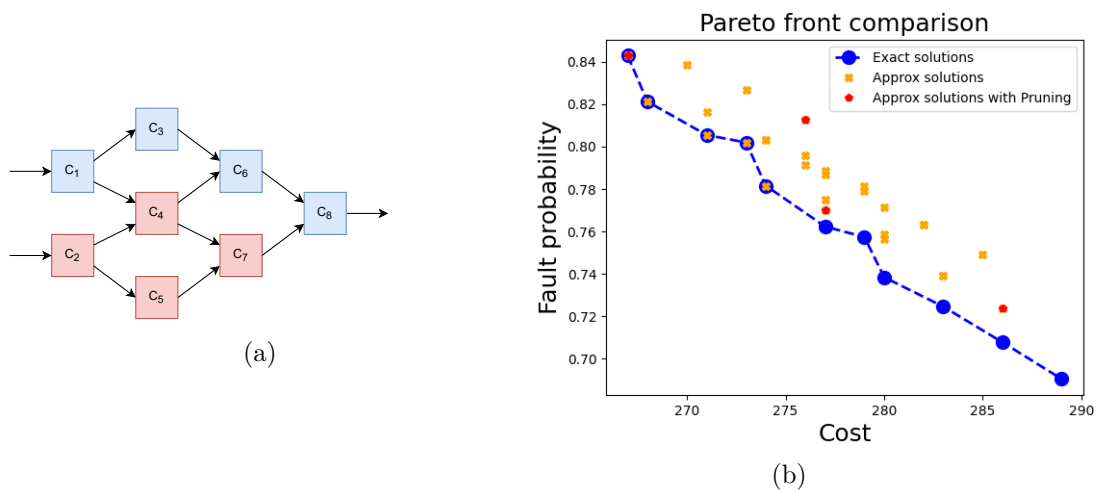


Figure 9.3: (a) Basic system composed of 8 components and 10 edges, (b) Pareto solutions of exact and approximate methods.

Compared to the results obtained with the exact method, we obtained twenty-eight approximate solutions, produced by four sub-solutions from Part 1 and seven sub-solutions from Part 2. We obtained only four solutions by applying the pruning rule with the following threshold:

$$th = avg_{cost} \pm (max_{cost} - avg_{cost})/2.$$

Table 9.1 lists the comparison of the solutions. Figure 9.4 shows the performance for the three methods employed. Compared to the exact method, execution time for the approximate method without pruning is higher as the

number of approximate solutions (retrieved by combining the two sets of sub-solutions) is higher. Instead, the memory consumption is lower as the partitioning reduces the size of the original problem.

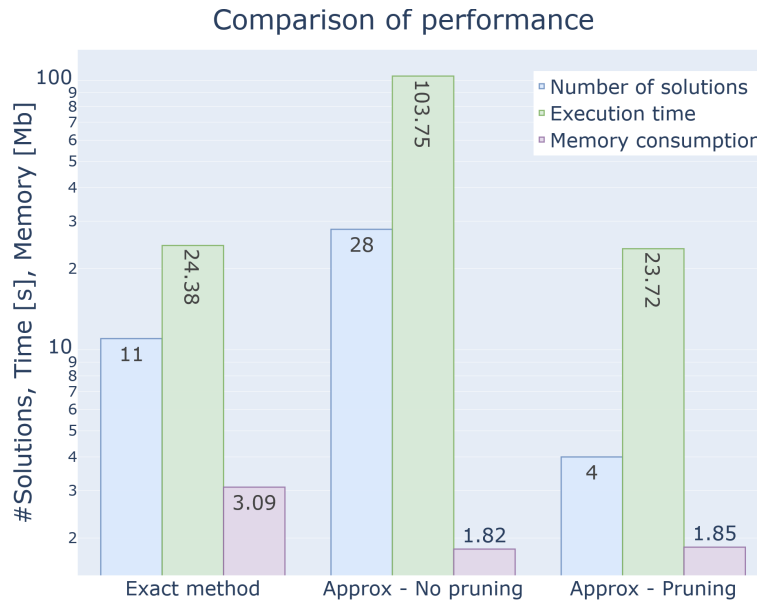


Figure 9.4: Comparison of performance of exact and approximate methods for a complex system of eight components.

Example system with 10 components

Recall the example system composed of ten components that we have introduced in Section 7.4.5, and consider the KL partitioning of Figure 9.5a. In order to highlight the influence of the pruning threshold on the number of the resulting solutions, on the execution time, and - as a consequence - on the quality of those solutions, we considered the following cases:

- Exact method
- Approximate method case 1: without pruning
- Approximate method case 2, pruning threshold set to:

$$th_1 = avg_{cost} + (max_{cost} - avg_{cost})/2$$

- Approximate method case 3, pruning threshold set to: $th_2 = avg_{cost}$

Table 9.1: Comparison of solutions for system in Figure 8.4a
(considering two objective functions).

Exact solutions		Approximate solutions		Approximate solutions with Pruning	
Fault probability	Cost	Fault probability	Cost	Fault probability	Cost
0.8428742271125870	267.0	0.8428742271125870	267.0	0.8428742271125870	267
-	-	0.8384519153632812	270.0	-	-
-	-	0.8268684883311485	273.0	-	-
0.8211849152731773	268.0	0.8211849152731773	268.0	-	-
-	-	0.8161521568936392	271.0	-	-
-	-	0.8126527681152138	276.0	0.8126527681152138	276
0.8054633288060601	271.0	-	-	-	-
-	-	0.8054633288060601	271.0	-	-
-	-	0.8029697779107720	274.0	-	-
-	-	0.8029697779107720	274.0	-	-
0.8019181711701844	273.0	0.8019181711701844	273.0	-	-
-	-	0.7959141452842944	276.0	-	-
-	-	0.7914096016987506	276.0	-	-
-	-	0.7885578430936907	277.0	-	-
-	-	0.7867917495189143	277.0	0.7699883880550559	277
0.7815761362114649	274.0	0.7815761362114649	274.0	-	-
-	-	0.7813215583627837	279.0	-	-
-	-	0.7791567833072470	279.0	-	-
-	-	0.7749555262341146	277.0	-	-
-	-	0.7711964597414617	280.0	-	-
-	-	0.7699883880550559	277.0	0.7699883880550559	277
-	-	0.7633660081939778	282.0	-	-
-	-	0.7633660081939778	282.0	-	-
0.7623721095138326	277.0	-	-	-	-
-	-	0.7588643520113397	280.0	-	-
0.7573673206150622	279.0	-	-	-	-
-	-	0.7564772651459938	280.0	-	-
-	-	0.7493645944397505	285.0	-	-
-	-	0.7390648546647709	283.0	-	-
-	-	0.7390648546647709	283.0	-	-
0.7384561893264687	280.0	-	-	-	-
0.7245808564878988	283.0	-	-	-	-
-	-	0.7236255642020005	286.0	0.7236255642020005	286
0.7077806036827384	286.0	-	-	-	-
0.6904902531138207	289.0	-	-	-	-

- Approximate method case 4, pruning threshold set to:

$$th_3 = avg_{cost} - (max_{cost} - avg_{cost})/2$$

Figure 9.6 shows the comparison of performance for the methods employed. Partitioning speeds up the optimization task because the more strict the threshold is, the more sub-solutions are discarded.

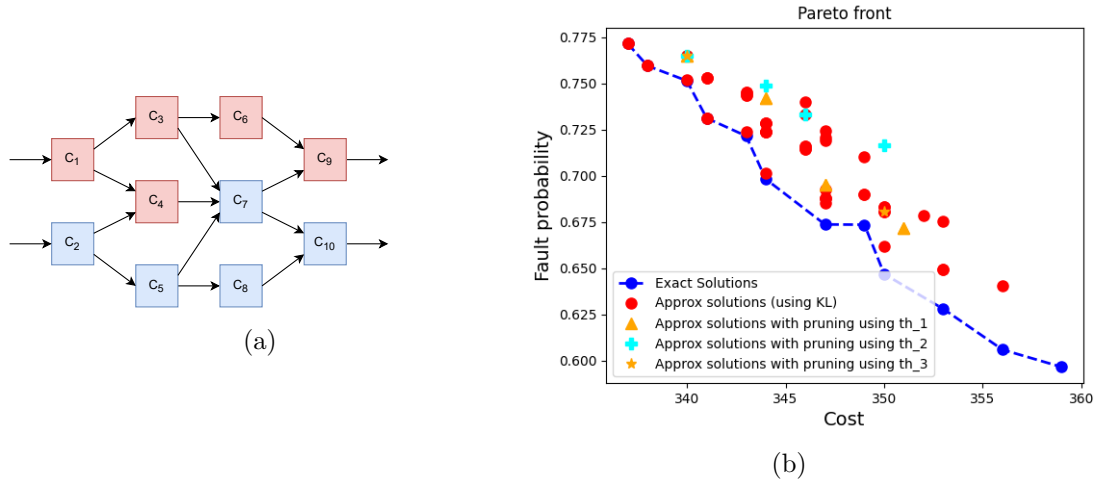


Figure 9.5: (a) Partitioning of a basic system composed of 10 components and 13 edges, (b) Pareto solutions of exact and approximate methods.

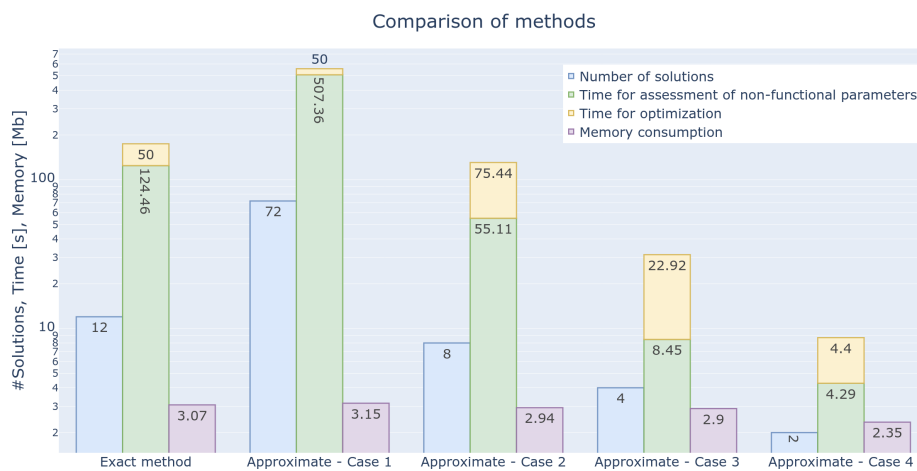


Figure 9.6: Comparison of performance of exact and approximate methods for a complex system of ten components.

Example system with 14 components

We did the same for the example system of fourteen components presented in Section 7.4.5, and considering the KL partitioning of Figure 9.7a.

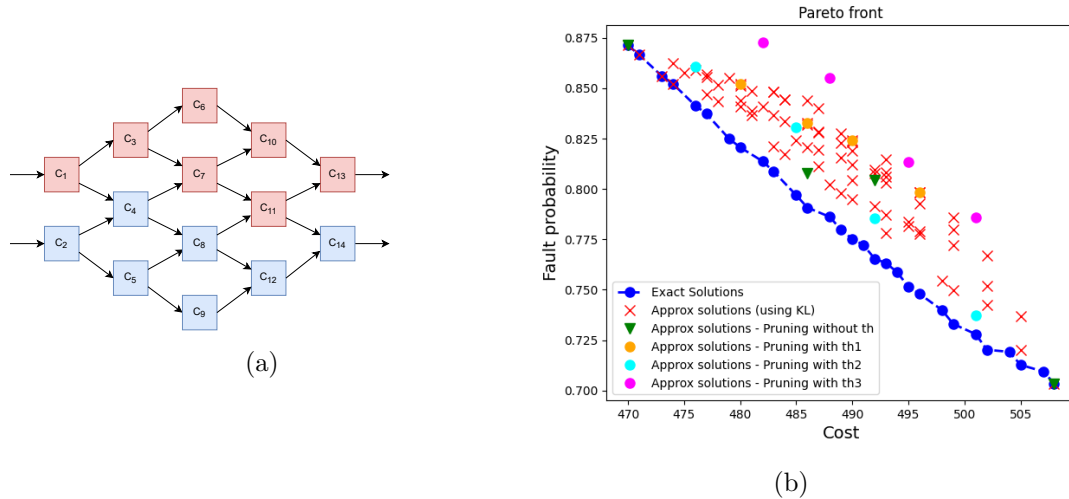


Figure 9.7: (a) Partitioning of a basic system composed of 14 components and 20 edges, (b) Pareto solutions of exact and approximate methods.

From Figures 9.8a and 9.8b, it is evident that partitioning drastically reduces time executions, but the more strict the threshold is, the more sub-solutions we discard, and the lower is the quality of the combined solutions. The bar diagram in Figure 9.8a reports the comparison between the exact method, the approximate method (without pruning), and the approximate method pruning all dominated solutions. The graph in Figure 9.8b considers the following cases:

- Exact method
- Approximate method case 1: pruning all dominated solutions (without using any threshold for solutions acceptance).
- Approximate method case 2, pruning threshold set to:

$$th_1 = avg_{cost} + (max_{cost} - avg_{cost})/2$$

- Approximate method case 3, pruning threshold set to: $th_2 = avg_{cost}$

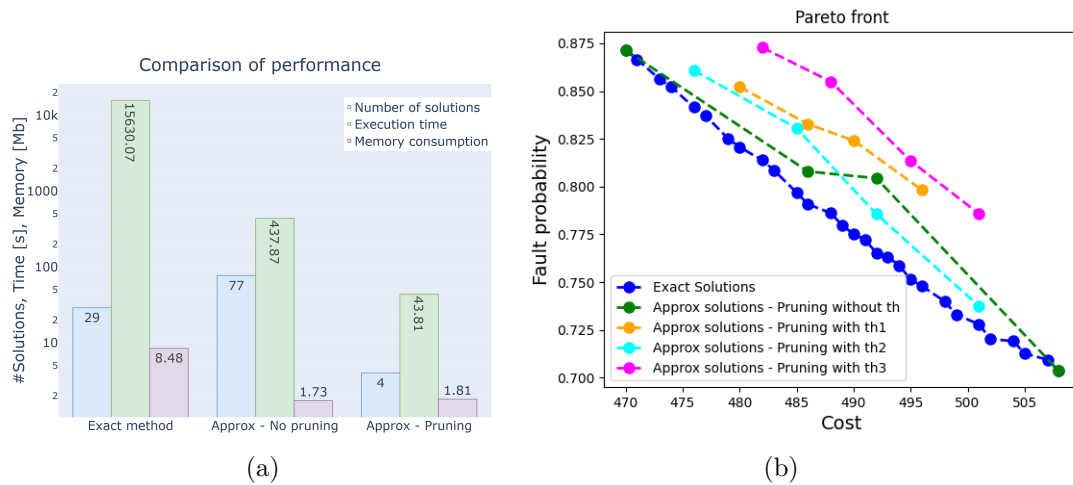


Figure 9.8: (a) Comparison of performance of exact and approximate methods for a complex system of fourteen components, (b) Approximate solutions move away from exact solutions as the pruning threshold is tighten up ($th_1 < th_2 < th_3$).

- Approximate method case 4, pruning threshold set to:

$$th_3 = avg_{cost} - (max_{cost} - avg_{cost})/2$$

Example system with 24 components

In this experiment, we performed a comparison between KL and Metis algorithms for partitioning the same given system. We considered an example system composed of twenty-four basic components, with two redundant patterns each, and two design objectives. We used the approximate method with pruning, executing two different runs to apply the two partitioning algorithms to the same example systems (see Figure 9.9 and Figure 9.10). This led to the very same solutions, as illustrated in Figure 9.11. Figure 9.12 illustrates that partitioning the systems in more parts improved time performance.

Please note that in this case partitioning the given system in two and three parts respectively led to the very same solutions, but in general this is not always true. That depends on the edge-cut of the specific partitioning, as illustrated in the next example, and detailed in Section 9.5.

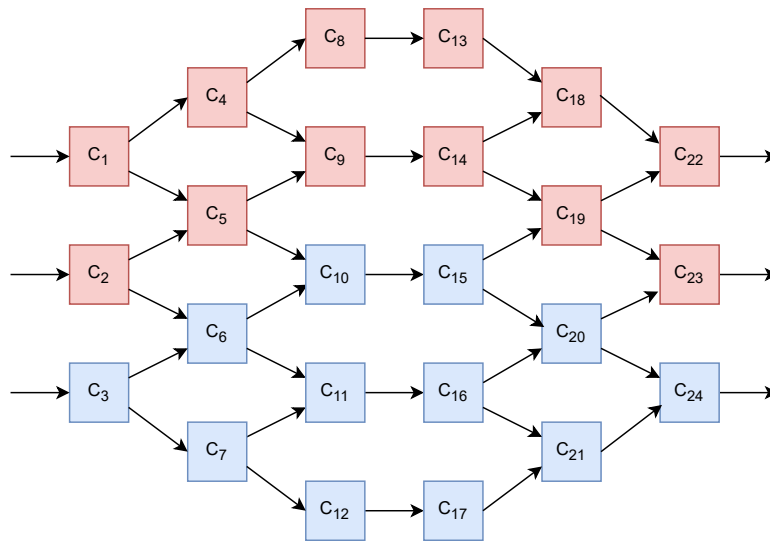


Figure 9.9: Partitioning of a system composed of 24 components and 33 edges using KL.

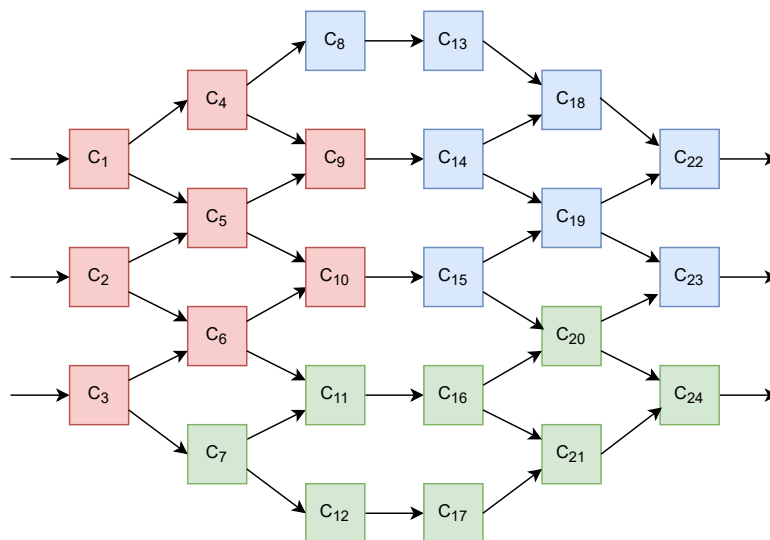


Figure 9.10: Partitioning of a system composed of 24 components and 33 edges using Metis.

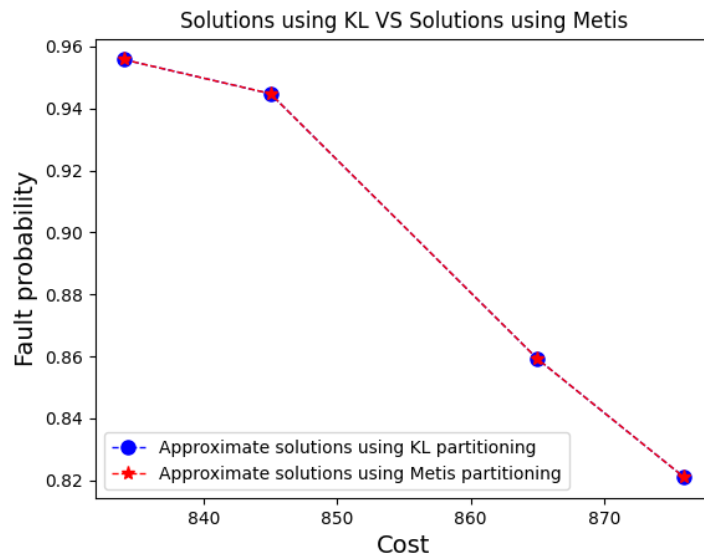


Figure 9.11: Comparison of KL (2 parts) and Metis (3 parts) algorithms for partitioning the example system of 24 components: the two algorithms led to the very same solutions.

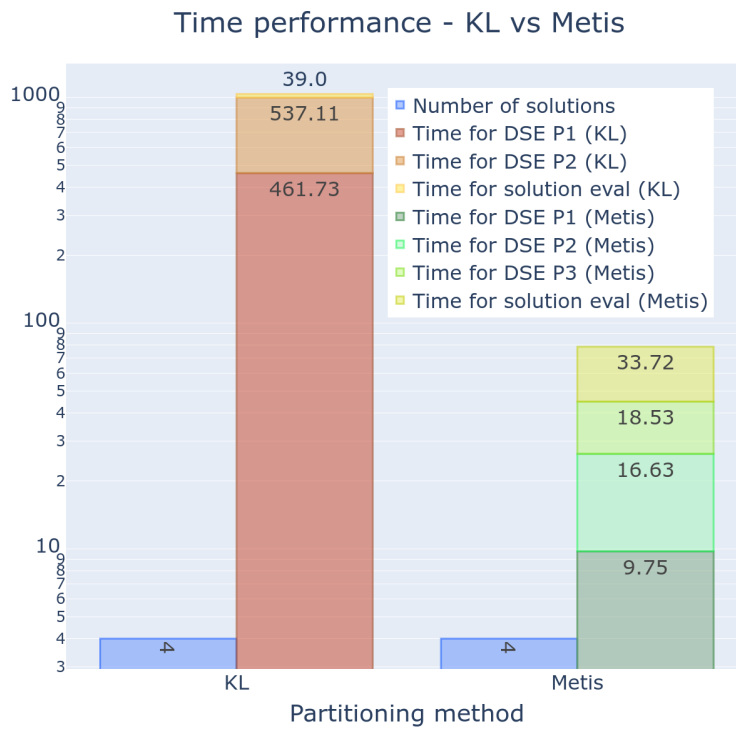


Figure 9.12: For large systems, partitioning in more sub-architectures improves time performance.

Example system with 69 components

With a system of sixty-nine components, we first tried to partition the system using the KL algorithm, then used Metis and partitioned the system into three, four, and eventually five parts. With complex architectures and large libraries of redundant patterns, we found experimentally that to obtain reasonable execution times we should limit the number of components for each part to around fourteen.

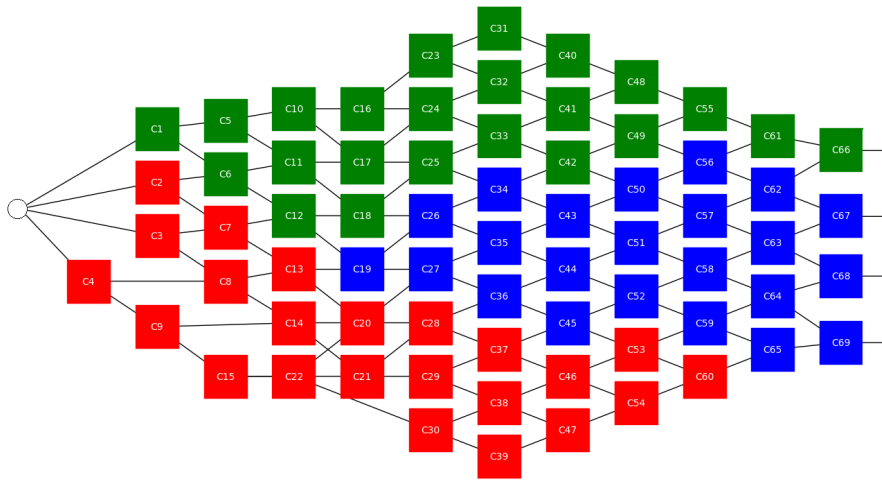


Figure 9.13: Partitioning of a system composed of 69 components and 120 edges using Metis (Number of parts = 3, Edge-cut = 17).

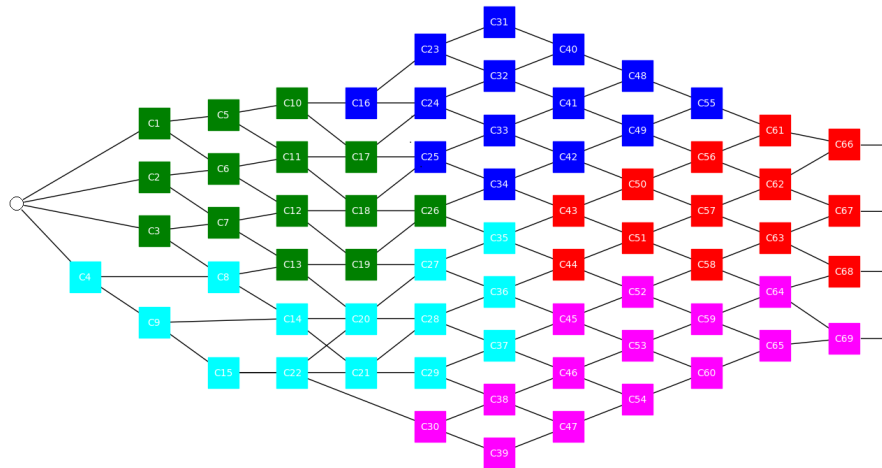


Figure 9.14: Partitioning of a system composed of 69 components and 120 edges using Metis (Number of parts = 5, Edge-cut = 41).

Partitioning the system into three parts (as illustrated in Figure 9.13) resulted into an edge-cut value of 17, while partitioning it into five parts (as illustrated in Figure 9.14) resulted into an edge-cut value of 41. Increasing the number of partitions is crucial to speed up the computation, but the higher the number of partitions, the higher the number of edge-cuts, the lower the quality of the solutions for the basic system. More details are provided in Section 9.5.

Strategy to determine the pruning threshold

As already stated in Sub-section 8.5, since the number of approximate solutions can increase rapidly, a pruning technique is advised, in order to reduce the size of the search area by removing less promising sections. The pruning and ranking methods basically select the most relevant sub-solutions that will be combined into near-optimal solutions for the original architecture. Figure 9.15 illustrates the concept described in Figure 8.3 for the example system of ten components of Figure 9.5. We applied the pruning strategy to the two sub-architectures resulting by partitioning the system with the KL algorithm. By tightening or widening the acceptance threshold, we exclude or include a set of sub-solutions, resulting in quicker computations but less accurate final solutions, or slower computations but more accurate final solutions. If setting a threshold, it should be not too much wide if you want to obtain advantages from it, but it should be not too much tight otherwise you can incur in the situation of Figure 9.15b, in which the threshold 1 (in red) is not suitable because it would prune every sub-solution.

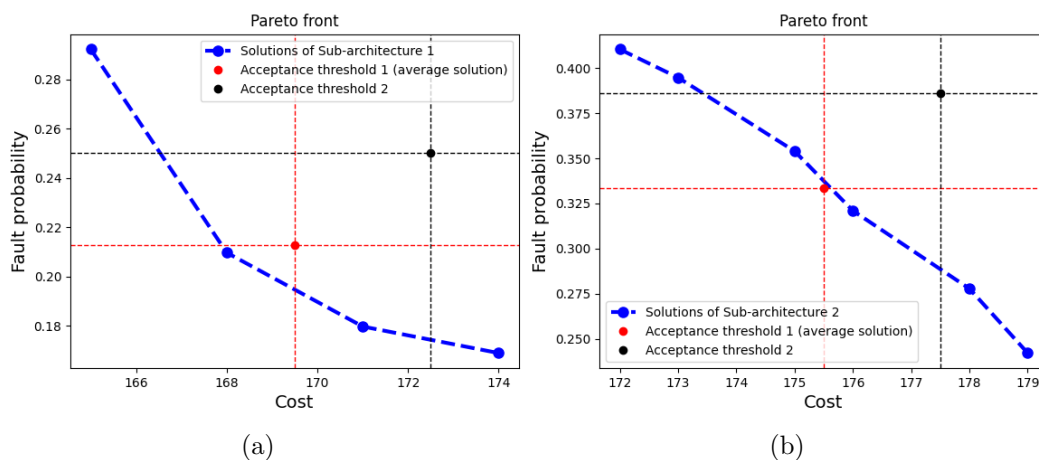


Figure 9.15: Pruning of sub-solutions of Part 1 (a) and Part 2 (b) for two different thresholds.

The threshold selection may be adapted from case to case, depending on the number of basic components, the size of redundant pattern library, and on the number and kind of optimization objectives. Please note that the more sub-solutions we have, the higher is the number of global solutions obtained by combining them. With very large systems, we determined the value of the threshold experimentally using the following procedure. First execute the algorithm without pruning, if the computation time is too slow or the memory usage too high, then perform the execution again lowering the threshold by incremental steps until you observe a reasonable computing time or a memory consumption that does not saturate the RAM memory, and allow you to obtain an adequate number of sub-solutions.

9.3 Results

The overall performance of the approximate method has three main contributions:

- the system partitioning,
- the application of the exact method to the individual partitions,
- the assembly of solutions for the original system and their evaluation.

Partitioning is performed using the KL algorithm to split the original graph in two partitions. We use instead Metis to have three or more partitions, which could be useful for very large systems. In both cases, the time required for partitioning is negligible compared to the time for DSE. The results indicated that the quality of the solutions is not affected by the number of partitions if the edge-cut is the same and if we do not apply any pruning strategy, as illustrated in Figure 9.11. However, the larger the system, the more convenient it is to divide it into multiple partitions. On one hand, the time for the DSE of various small partitions is significantly more restrained compared to that of few large partitions. On the other hand, the higher the number of partitions, the higher the edge-cut value. See Section 9.5 for more details.

Regarding the performance of the application of the exact method to the partitions, it has been analyzed extensively in Chapter 7. Obviously, we need a call to the solver for each partition, but the sub-problems are much less complex than the original problem.

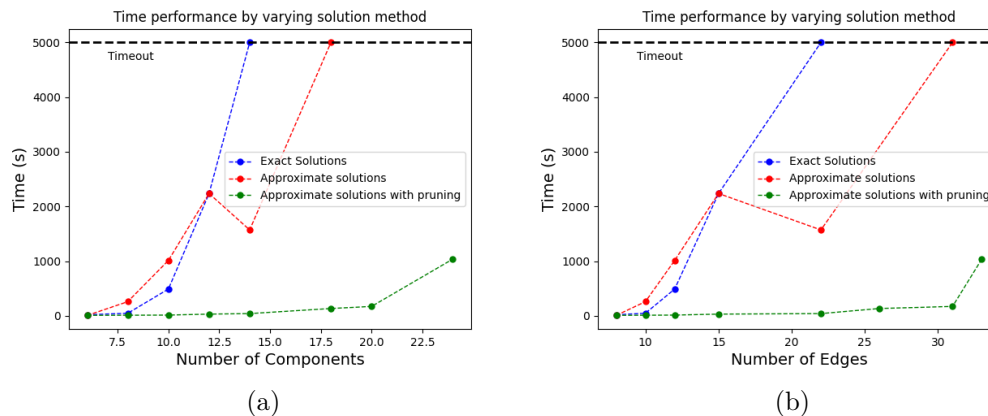


Figure 9.16: Comparison of time performance of solution methods by varying the number of nodes (a) and the number of edges (b). Note: there is a bump in the curve of approximate solutions because for systems composed by 14 components onwards we employed graph partitioning.

As for partitioning, also combining solutions is a negligible task, since it is a simple flattening process. Conversely, their evaluation is another burdensome task, since we recall the solver for each solution. The results indicate that the number of approximate solutions grows exponentially with the number of components and that of partitions (see Figure 9.16). For complex architectures with more than thirty components and more than three redundant patterns for each basic component, the pruning strategy with an appropriate acceptance threshold that speeds up the search is therefore fundamental. As already stated above, partitioning drastically reduces time executions, but the more strict the threshold is, the more sub-solutions we discard, and the lower is the quality of the combined solutions, as illustrated in Figure 9.8b. If we consider all possible combinations of sub-solutions from partitions, we will obtain a set of solutions that will surely include the exact solutions computed without partitioning. Using an acceptance threshold will make we miss some of the exact solutions. And it will be more likely to happen when the tighter the threshold is.

9.4 Test problem

In order to compare our method with other existing methods, we explored a test problem introduced by Fyffe et al. [178] that is widely used in the reliability optimization literature [179, 311, 312, 313, 185, 314, 315, 316, 317, 318, 319,

320, 321, 322, 323, 324, 325]. It is a S-P system with fourteen components connected in series, each with three or four component choices for redundancy. The failure probability of each unit in the system is assumed to be independent. Given the total amount of 130 units of system cost and 170 units of system weight constraint, the problem is to find the optimum number of redundancy to use and the optimum design alternative of each stage that will result in the greatest system reliability while keeping the total cost and weight less than the given amount. Table 9.2 reports the input data for the given problem. Each component is characterized by a reliability (R), a cost (C), and a weight (W), as reported in Table 9.2.

Table 9.2: Input data for the test problem

Component	Design alternative											
	1			2			3			4		
	R	C	W	R	C	W	R	C	W	R	C	W
1	0.90	1	3	0.93	1	4	0.91	2	2	0.95	2	5
2	0.95	2	8	0.94	1	10	0.93	1	9	-	-	-
3	0.85	2	7	0.90	3	5	0.87	1	6	0.92	4	4
4	0.83	3	5	0.87	4	6	0.85	5	4	-	-	-
5	0.94	2	4	0.93	2	3	0.95	3	5	-	-	-
6	0.99	3	5	0.98	3	4	0.97	2	5	0.96	2	4
7	0.91	4	7	0.92	4	8	0.94	5	9	-	-	-
8	0.81	3	4	0.90	5	7	0.91	6	6	-	-	-
9	0.97	2	8	0.99	3	9	0.96	4	7	0.91	3	8
10	0.83	4	6	0.85	4	5	0.90	5	6	-	-	-
11	0.94	3	5	0.95	4	6	0.96	5	6	-	-	-
12	0.79	2	4	0.82	3	5	0.85	4	6	0.90	5	7
13	0.98	2	5	0.99	3	5	0.97	2	6	-	-	-
14	0.90	4	6	0.92	4	7	0.95	5	6	0.99	6	9

The problem can be generalized as follows (see Figure 9.17). The system has a total of s sub-systems arranged in series. For each sub-system, there are n functionally equivalent components arranged in parallel. Each component has three different parameters: cost, weight, and reliability. The n components are to be selected from m available component types, where multiple copies of each type can be selected.

To compare the results we had to perform a slight modification to our formalism. In Section 5.1, we assumed - among other things - to consider for each component only one redundant pattern for redundancy. To face the

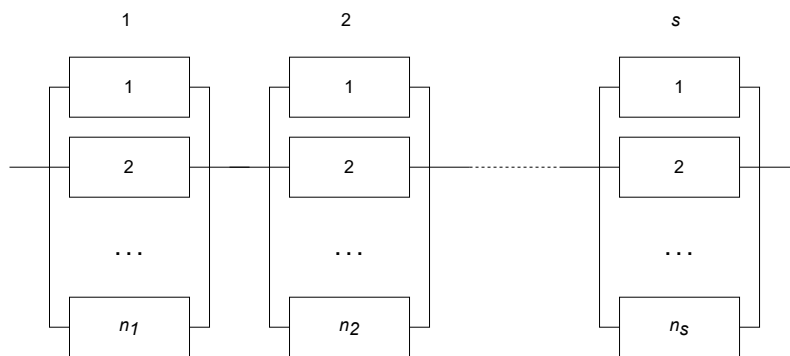


Figure 9.17: Structure of the S-P system of the test problem

test problem, we considered the computing modules of the redundant patterns employed as redundant alternatives in parallel, and set the fault probability of comparators and voters to zero. The redundant patterns employed were: CMP, TMR, and 3oo4. Furthermore, since in some works the authors also model an imperfect switching mechanism with a switch reliability set to 0.99, we performed an additional execution with fault probability of comparators and voters $F_C/F_V = 0,01$, in place of the switch.

For instance, the first design alternative for the component 1 has the following cost values:

$$M_1 \rightarrow R = 0.90, C = 1, W = 3$$

And the second design alternative for the component 1 has the following cost values:

$$M_2 \rightarrow R = 0.93, C = 1, W = 4$$

This means that, according to our formalism, the example configuration in Figure 9.18 translates in the configuration of Figure 9.19. Moreover, since the test problem states that for each stage only one design alternative should be used, we had to exclude component mixing, i.e., we had to use identical modules for each stage (see Figure 9.20).

With our formalism, the given optimization problem has a basic system of fourteen components and a global pattern library of one hundred twenty redundant patterns. In order to obtain a reasonable time performance, we split the original system into four partitions, as illustrated in Figure 9.21. Our method found a total number of 334.805 exact solutions satisfying the requirements, resulting from the combination of the sub-solutions reported in Table 9.3. The solution with highest reliability using ideal comparators/voters resulted the following:

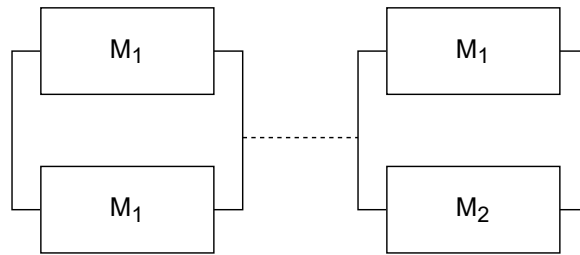


Figure 9.18: Each stage of the redundant architecture is composed by one to four identical modules in parallel

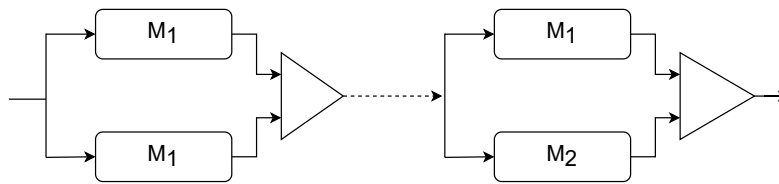


Figure 9.19: With our formalism, we used CMP, TMR, and M-oo-N patterns to face the test problem.

Reliability, Weight, Cost = [0.998849923, 170, 112].

The solution with highest reliability using comparators/voters with $R = 0,99$ (i.e., $F_{prob} = 0,01$) resulted the following:

Reliability, Weight, Cost = [0.9977464147, 170, 112].

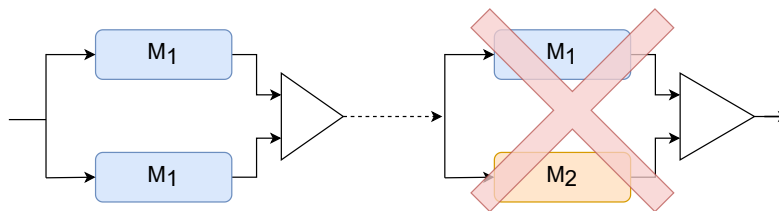


Figure 9.20: Component mixing is not allowed in the test problem, i.e., we have to use identical modules for each stage of the architecture.



Figure 9.21: Partitioning for the test problem system

Table 9.3: Sub-solutions for the test problem.

Part	Solutions	Execution Time [s]	Memory Usage [Mb]
1	140	67.85587217699958	2.4559555053710938
2	65	27.189838600001167	2.466968536376953
3	29	3.693340691999765	2.468883514404297
4	37	13.103370544999052	2.47003173828125

Figure 9.23 illustrates a comparison of our best solution with those of related works. Please note: only Reliability and Cost are reported, as all those solutions have Weight=170.

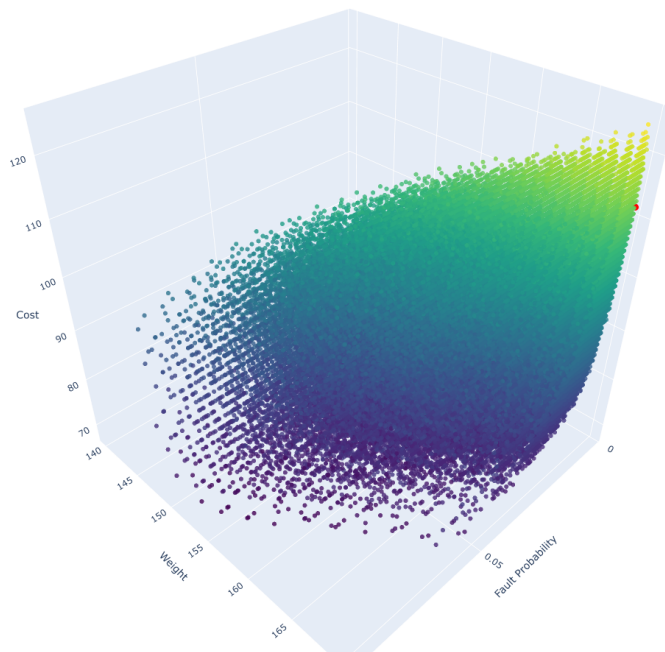


Figure 9.22: Solutions for the test problem system. In red, the one with highest reliability.

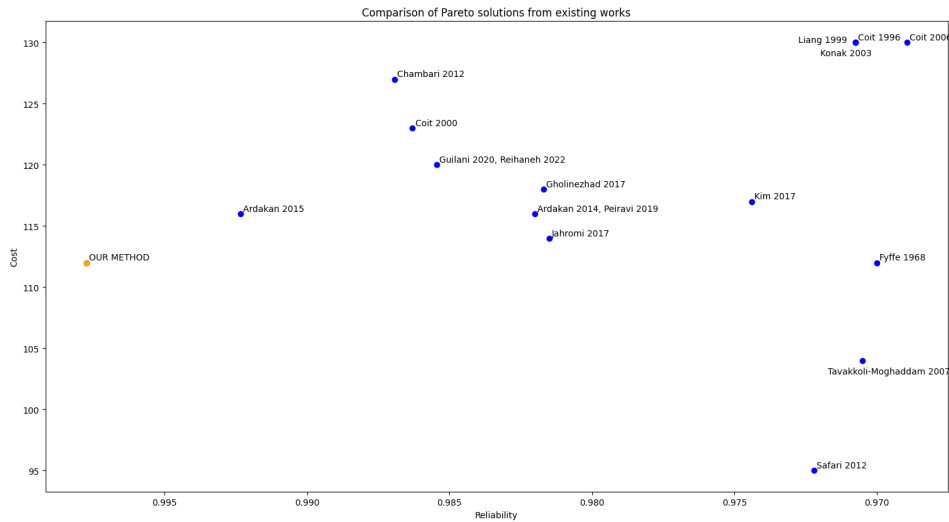


Figure 9.23: Comparison of solutions with related works

In order to figure out if we could improve the solution, we repeated the experiment extending the library of patterns to include component mixing scenarios. At first run, the solution with highest reliability allowing component mixing and using comparators/voters with $R = 0,99$ (i.e., $F_{prob} = 0,01$) was the following:

$$\text{Reliability, Weight, Cost} = [0.998849923, 170, 104].$$

This means that the employment of component mixing allowed us to improve cost and reliability, still meeting the requirement on weight. Figure 9.24 shows the comparison of the solutions with highest reliability found with and without component mixing. Figure 9.25 shows the comparison with related works, with reference to reliability and cost objectives. The solution found with our method using component mixing had higher reliability and lower cost compared to the solution found with our method not using component mixing. In any case, at best of our knowledge, both solutions outperform all the other existing methods.

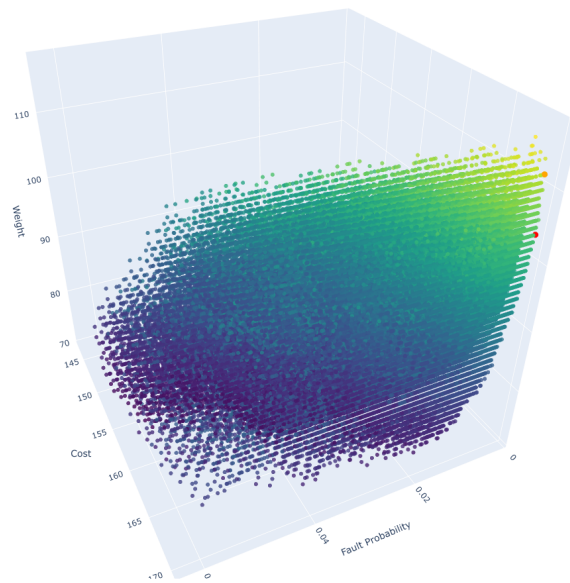


Figure 9.24: Solutions for the test problem system. In red, the solution with highest reliability allowing component mixing, in orange the solution obtained without component mixing, illustrated in Figure 9.22.

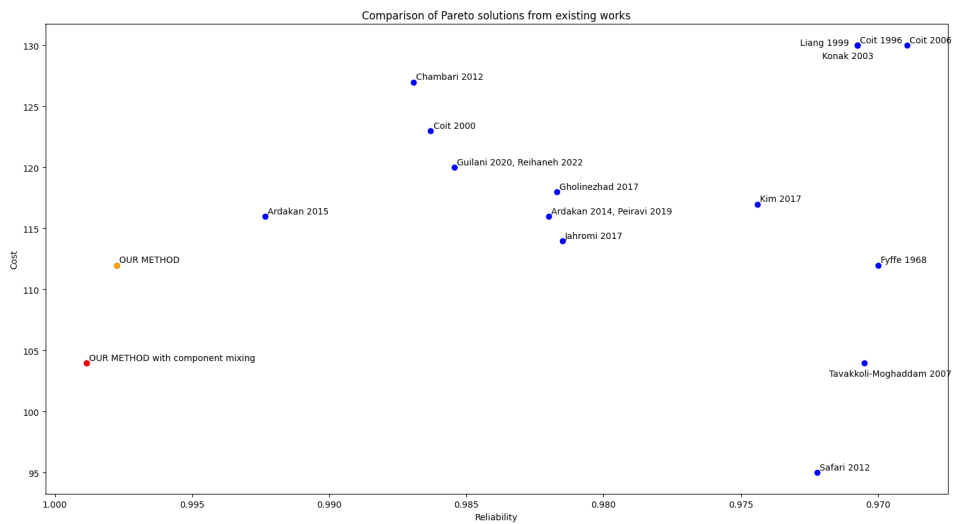


Figure 9.25: Comparison of solutions with related works. In red, the solution found by our method with component mixing and applying partitioning with a balanced pruning strategy.

Please note that using component mixing, according to our formalism, led to a library of three hundred and fifty-two patterns. To face this problem, we used again graph partitioning. Furthermore, we had to prune some solutions. Without pruning, the total number of solutions found with our method was 16.770.539.520. Anyway, our computing system ran out of memory when evaluating all those solution. With pruning, we reduced the total number of solutions to 1.523.712. Hence, there might still be the chance that we missed some optimal solutions. We repeated the execution again, varying the pruning strategy in order to favor sub-solutions with higher reliability, and obtain as solution with highest reliability the following one:

$$\text{Reliability, Weight, Cost} = [0.999716964 \ 167 \ 95].$$

Figure 9.26 illustrates the solutions found by our method, described above. In orange, the best solution found by our method without component mixing, as requested by the test problem. In red, the solution found by our method with component mixing and applying partitioning with a balanced pruning strategy for all design objectives (the same for each sub-architecture). In purple, the best solution found by our method so far, by applying partitioning with a pruning strategy (specific for each sub-architecture) designed to favor sub-solutions with higher reliability and lower cost. Figure 9.27 illustrates the comparison of our solutions with those of related works.

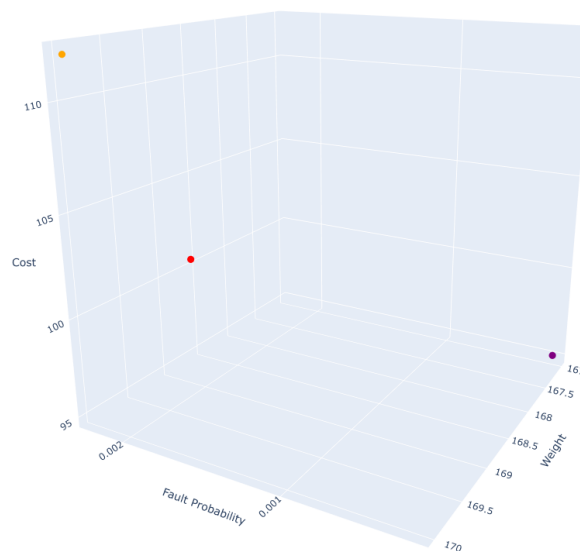


Figure 9.26: Solutions with higher reliability found by our method.

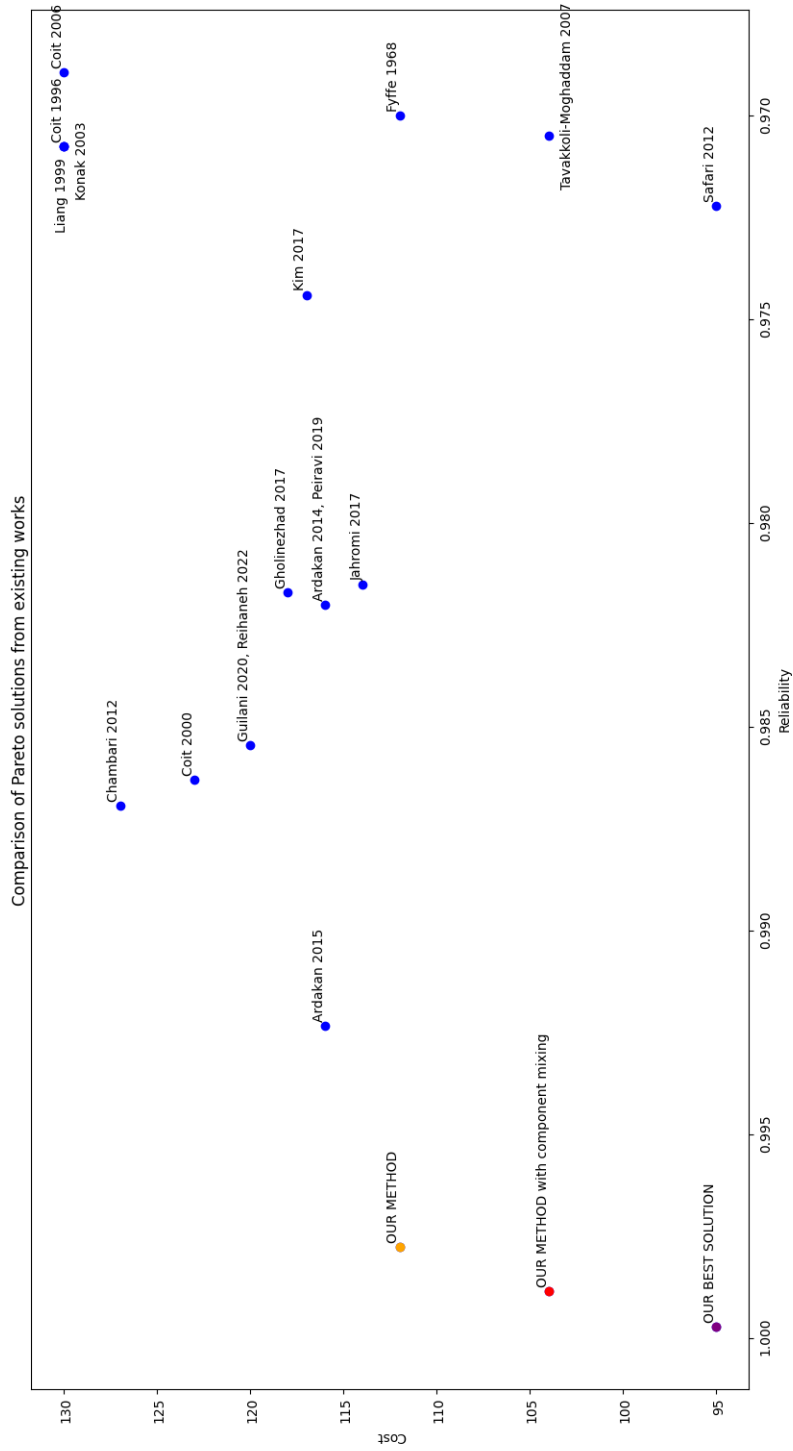
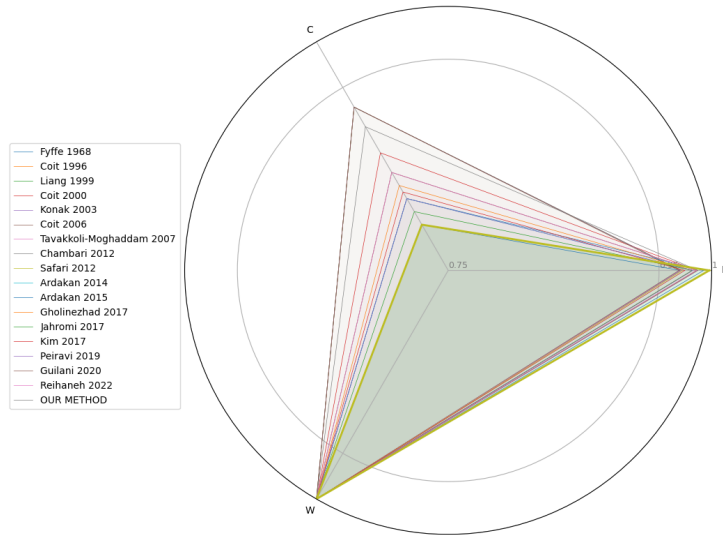
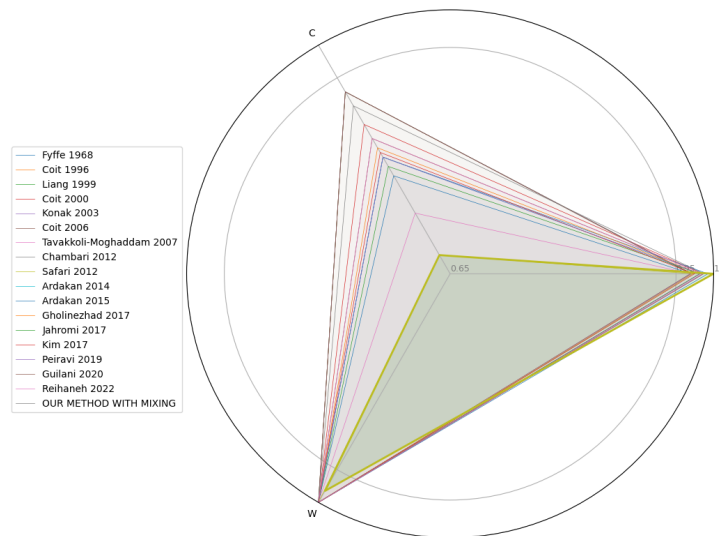


Figure 9.27: Comparison of solutions with related works. In purple, the best solution found by our method, by applying partitioning with a pruning strategy designed to favor sub-solutions with higher reliability and lower cost, specific for each sub-architecture.

Figure 9.28a and Figure 9.28 illustrate the comparison of our solutions (without and with component mixing) with those of related work in terms of Kyviat diagrams.



(a)



(b)

Figure 9.28: Comparison of our solution with related works: (a) without component mixing, (b) with component mixing .

9.5 Applicability and Limitations

The approximate method allows us to face problems that are prohibitive for the exact method, such as complex architectures with dozens of components and patterns. Nevertheless, there are a couple of drawbacks.

First, the efficiency of the method is highly dependent on the partitioning method, which aims at reducing the edge-cut. Because of cyclic partitions in the basic systems, there could be different architectures leading to the same partitioning, resulting in a wrong approximation of the final solutions. For instance, the two architectures in Fig.9.29a and Fig.9.29b are both composed of seven components, but the second one has some edges that connect components in a way that lead to cyclic partitions, causing a high edge-cut. How the system is partitioned is therefore a key point.

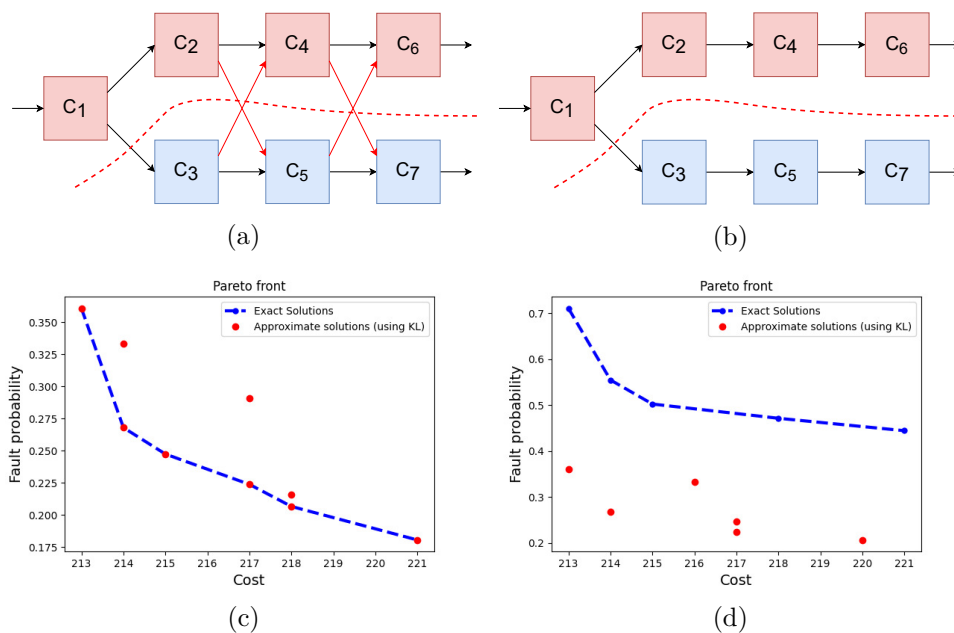


Figure 9.29: Example of two architectures producing the same partitionings, without (a) and with (b) cyclic interdependencies among components, resulting in good (c) and bad (d) approximate solutions.

A second issue can arise when partitions include components that are not connected in the basic system. Common partitioning algorithms aim at reducing the edge-cut, without checking if the nodes included in the same partition are connected or not. As a consequence, a scenario like the one depicted in

Fig.9.30 could happen. To prevent that situation, we employed partitioning algorithms with weight functions on nodes and edges, which means minimize the total communications volume, i.e., the sum of node/edge weights cut. With reference to our specific problem, those weights must be related to the reliability of the individual components. For example, among our tests, we assigned a greater weight to nodes with higher probability of fault or to the edges connecting those nodes. In other words, our aim was to split the original system into two or more sub-systems while minimizing dependencies from the point of view of reliability. However, this does not solve the problem. To split the system in partitions including connected components only, we should modify the partitioning algorithm to express such constraint. We reserve this for our future work.

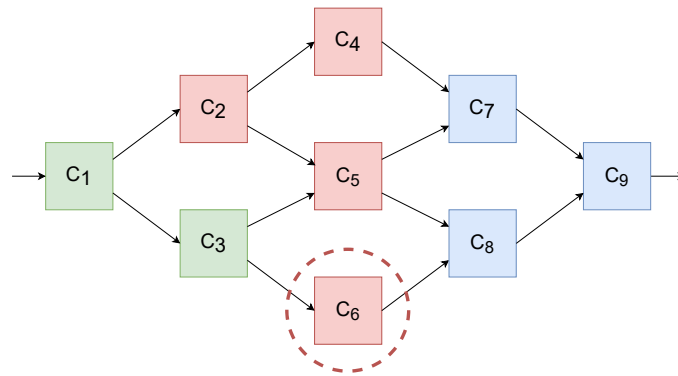


Figure 9.30: The sub-architecture in red includes components not connected in the original architecture.

Chapter 10

Conclusions and Future Work

The increasing demands on ES functionality make their design a challenging issue that requires design automation. Given that many ES are enclosed within safety-critical systems, reliability becomes mandatory to be targeted at design time. In this thesis, we aim at optimizing a system design with respect to multiple objectives with a special focus on reliability as a fundamental design objective. In this chapter, we summarize the main contributions of this work followed by a perspective on future work.

10.1 Summary

When it comes to creating reliable systems, redundancy plays a fundamental role. In this work we presented an approach to the automated analysis of redundancy architectures, which is able to produce a symbolic representation of the reliability function and other non-functional requirements of an ES architecture from the architectural description. We used this symbolic approach to encode the RAP, leveraging the power of SMT to systematically explore the design space, allowing us to deal with a set of architectures at the same time, in front of multiple design objectives, and thus, easing the way to redundant architecture synthesis. In this approach, a system architecture is modeled by

a DAG, and each computing element has a set of redundant patterns that can improve the reliability (and affect other non-functional parameters). Each system alternative design explored by the DSE process is specified by a list of allocated patterns resources. The process relies on FM in different phases: we model the architecture using SMT(EUF), we adopt a Miter construction to describe the occurrence of a system failure, we reduce the problem of extracting the set of relevant fault configurations by reduction to an AllSMT problem, we leverage predicate abstraction for the efficient extraction of the CSs, we use BDDs to extract the symbolic functions. The proposed method scales very well, and allows us to analyze configurations of realistic size. One of the main contributions is the novel assessment of reliability. Another relevant contribution is the introduction of the approximate method that can help solve the optimization problem when enumerating every point is prohibitive. Furthermore, we introduced some refinements and smart solutions that improved further the performance, like binary encoding of configurations, halving of fault variables, implementation of different variable orderings for the BDD construction, and a pruning strategy for very large design spaces.

Chapter 5 is mainly dedicated to how the reliability function of a system is derived automatically. Chapter 6 presents how to obtain an automated and optimal allocation of redundant component instances. Chapter 7 presents case studies and experimental data to support the validity and efficiency of the method. Chapter 8 proposes a meta-heuristic optimization approach to deal with large optimization problems, in order to find a solution more quickly. Chapter 9 presents experimental data to support the validity and efficiency of the proposed method, and a test problem to compare it with other existing methods.

10.2 Models Assumptions, Limitations, and Applicability

In this work we made some assumptions, a detailed list is reported in Section 5.1. Among these, we assumed that there is no dependence between the components, and that each component is critical, meaning that its failure triggers the TLE. With this assumption, we can define the formula of the reliability of the system as a function of that of the individual components. Hence, we can provide the solver with the individual reliabilities (in terms of probability

of failure) and the formula for the overall reliability of the system, and then rely on it for the solutions. However, if the failure behavior of a component is affected in any way by the previous component being executed, or by the interface between them, these assumptions are no longer acceptable. For example, with non-homogeneous patterns (a TMR with 1 input and 3 outputs) the hypothesis of independence would fail.

We also assumed to refer to a coherent (or monotone) model. For coherent models, the definition of MCS corresponds with the formal notion of *Prime Implicants* (PI). For non-coherent models, MCS and PI differ for the latter contains negative literals while the former does not. In practice, a negative literal means that a component recovers from failure. We are excluding this scenario: if a component is faulty, it cannot recover from failure, and adding more failure events maintains the failure condition of the system.

The benefit of the architecture-based approach to reliability evaluation is evident, but it could add complexity to the models. A first trade off is in deciding when switching from exact to approximate method. From results presented above, this choice is based on the topology of the system under study, its size, the pattern library available, and the design objectives and constraints.

Another key trade off is in defining the number of partitions for decomposition of very large systems. Too many small partitions could lead to a large state space that may pose difficulties to its exploration or to a high edge-cut, resulting in missed solutions or wrong approximations. On the other hand, too few partitions may cause too long computing times and excessive resource usage, leading to abortion.

Having said that, the proposed method is suitable to reliability modeling of parametric families of ES architectures, is capable of exploring the design space in an efficient way, and scales very well, allowing us to analyze configurations of realistic size.

10.3 Exact or Approximate Method?

An efficient DSE basically requires a good exploration algorithm and automatic evaluation methods to flexibly evaluate the design objectives. With large MOOPs, because of the high complexity of finding Pareto optimal solutions and their usually very large number, the exact solution is often very

difficult, which motivates the study of approximation algorithms. Also in our case, we can address the RAP either exactly or approximately to find a set of optimal or sub-optimal solutions. The exact approach has the advantage of accuracy, but often takes too long for large-scale problems or even worst leads to abortion because of high resource usage. On the other hand, the approximate approach may solve those problems in a reasonable time, but suffers from sensitivity to parameter settings and lower accuracy with missed optimal solutions.

In general, in order to decide whether to use exact or approximate method, it is important to understand the trade-off between the resources and time required by exact approaches versus the difficulty in searching for appropriate parameters and the risk of missing relevant solutions when using approximate techniques. Experimentally, we found that for linear and rectangular architectures, the exact method can be applied even for systems composed of hundreds of components, if the pattern library is contained. Instead, for complex systems composed of dozens of components the resource usage is excessive. In this case, the approximate method is preferable to the exact one. More precisely, the decision is imposed as a result of the number of components of the basic system architecture, their connections, the size of the pattern library, and the number of design objectives.

10.4 Application to real systems

To give a glimpse of the power of the approach proposed, we present a real system case study and show how our model could be applied to it, although without providing real fault analysis results. Figure 10.1 illustrates an electric power system for power generation and distribution in a passenger aircraft [326, 327, 62, 170]. The main components are generators, contactors, buses, and loads. One or more supervisory control units actuate a set of electromechanical switches to dynamically distribute power from generators to loads, while satisfying safety, reliability, and real-time performance requirements. Buses deliver power to a number of loads. Contactors are electromechanical switches that connect components, and therefore determine the power flow from sources to loads. More in details, the system is divided into left and right parts. The corresponding generators (L-GEN and R-GEN) and Auxiliary Power Units (APUs) can power both parts. Components can be further

classified as High-Voltage (HV) and Low-Voltage (LV). Rectifier Units (RU) are used to convert AC power to DC power, while HV levels can be converted into LV levels using a Transformer-Rectifier Unit (TRU).

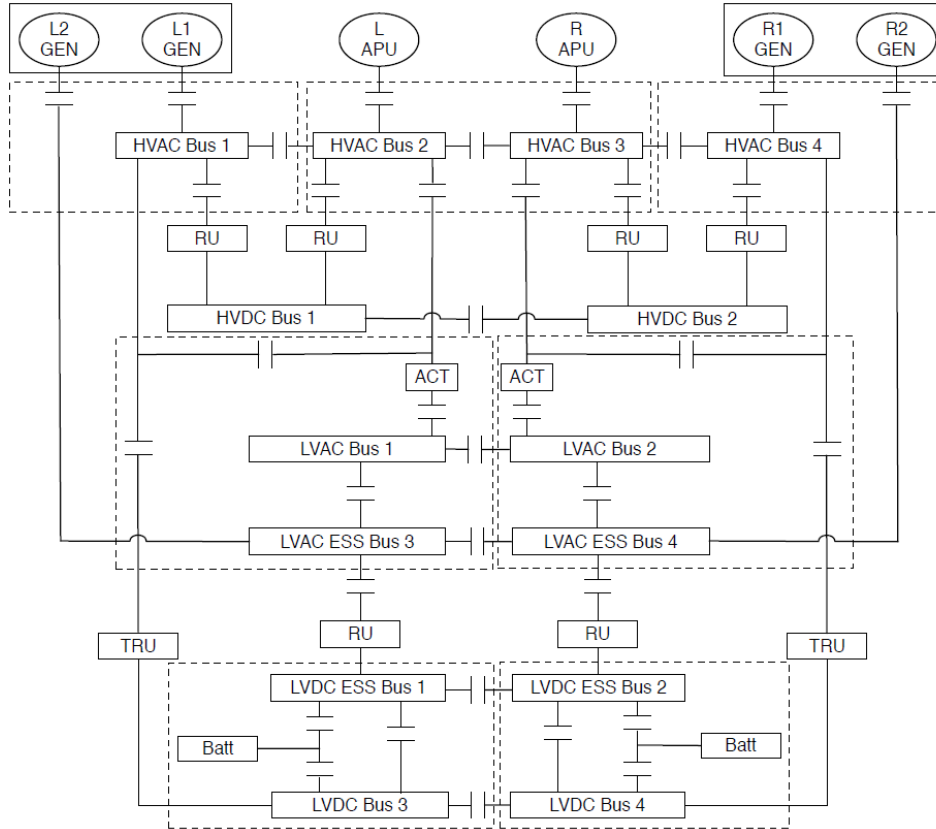


Figure 10.1: Power distribution network of an aircraft.

We can define the given system in terms of a graph structure in which both the components (nodes) and interconnections (edges) may fail. Both nodes and edges represent system components. In our formulation, the design consists of an interconnection of components taken from a library. In practice, every node and edge can be mapped to a library element that implements it. In other words, a component can be seen as an abstraction representing an element of the design, characterized by the following component attributes that are used to capture its properties:

- A set of input and output variables.
- A set of configuration parameters.
- A set of input and output ports for connection with other components.
- A set of behaviors, which can be generic functions.

- A set of non-functional models that allow computing non-functional properties of a component corresponding to particular valuations of its input variables and configuration parameters, including a cost and a failure probability (reliability model).

We could therefore create a library with the components mentioned above (generators, buses, and loads), while contactors could be modeled with edges. In a first approximation, we could assume that contactors and loads have no failures, while using a given fail probability for the other components. To sum up, we could produce a graph-based representation of the given system, and our approach could be very effective to achieve an optimal implementation that for example maximizes the reliability and minimizes a cost function.

10.5 Remarks

In this thesis, the following results have been accomplished. We designed a DSE process in which design alternatives are evaluated to *optimize* the architecture. We built a library of redundancy architectures applicable to the basic components, dealing with the complexities related to their selection and connections. We introduced a novel assessment of reliability which is part of the evaluation of alternative designs. In addition, we also considered other non-functional parameters, facing therefore a MOOP. Lastly, we also integrated the exact method with a heuristic that can help solve the optimization problem when the size of the design space makes enumerating every design point prohibitive. This approximate method is very flexible and adaptable to time and resources available. Besides, we introduced some refinements and smart solutions in every step of our method, in order to further improve the performance, such as binary encoding of configurations, fault variable sharing in redundant components, implementation of different variable orderings for the BDD, and a caching mechanism. We heavily relied on FM: architecture modeling was performed completely using SMT, we adopted a Miter construction to describe the occurrence of a system failure, we reduced the problem of extracting the set of relevant fault configurations by reduction to an AllSMT problem, we leveraged predicate abstraction for the efficient extraction of the CSs, we used BDDs to extract the symbolic functions. We used graph partitioning for facing very large problem instances. A key point worth to recall is that the symbolic

approach is strongly parallelizable, and, thus, our method could run efficiently on modern computing systems that offer more computational power.

10.6 Future work

Several topics have been taken into account to face the RAP, nevertheless there is still room for improvement. We are currently working on the following tasks:

- Investigating alternative and more efficient methods for the reliability assessment starting with the BDD of an instantiated architecture.
- Improving the strategy for sub-architectures pruning, and evaluating also additional strategies.
- In case of a high number of redundant patterns, investigating the pruning of the pattern library.
- Employment of different solvers for optimization, in order to perform a comparison.

In future work, we would also like to modify the partitioning algorithm to solve the issue illustrated in Fig. 9.30, and validate our method on real use case scenarios, like the one presented in Section 10.4. Successively, we will investigate the case where faults are associated with dynamics, going beyond combinatorial problems.

References

- [1] P. Derler, E. A. Lee, and A. L. Sangiovanni-Vincentelli, “Modeling cyber-physical systems,” *Proceedings of the IEEE*, vol. 100, no. 1, pp. 13–28, 2012. [Cited on page 1]
- [2] A. Burns and R. Davis, “Mixed criticality systems-a review,” *Department of Computer Science, University of York, Tech. Rep.*, pp. 1–69, 2013. [Cited on page 2]
- [3] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, “Assessing the state-of-practice of model-based engineering in the embedded systems domain,” in *International conference on model driven engineering languages and systems*, pp. 166–182, Springer, 2014. [Cited on page 2]
- [4] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004. [Cited on pages 6 and 19]
- [5] M. R. Lyu *et al.*, *Handbook of software reliability engineering*, vol. 222. IEEE computer society press CA, 1996. [Cited on page 6]
- [6] M. Trapp, D. Schneider, and P. Liggesmeyer, “A safety roadmap to cyber-physical systems,” in *Perspectives on the future of software engineering*, pp. 81–94, Springer, 2013. [Cited on pages xi, 7, and 8]
- [7] A. Birolini, *Reliability engineering: theory and practice*. Springer Science & Business Media, 2013. [Cited on page 8]
- [8] K. Schneider, *Verification of reactive systems: formal methods and algorithms*. Springer Science & Business Media, 2013. [Cited on pages 11, 21, 50, and 51]
- [9] J. Thomson, *High integrity systems and safety management in hazardous industries*. Butterworth-Heinemann, 2015. [Cited on page 12]

-
- [10] W. F. Larsen, “Fault tree analysis,” tech. rep., PICATINNY ARSENAL DOVER NJ, 1974. [Cited on page 14]
- [11] C. A. Ericson and C. Ll, “Fault tree analysis,” in *System Safety Conference, Orlando, Florida*, vol. 1, pp. 1–9, 1999. [Cited on page 14]
- [12] M. Bozzano and A. Villaflorita, *Design and safety assessment of critical systems*. Auerbach Publications, 2010. [Cited on page 14]
- [13] B. IEC, “60812: 2018 bsi: Failure modes and effects analysis (fmea and fmeca),” *British Standards Institution*, 2018. [Cited on page 17]
- [14] I. E. Commission *et al.*, “Iec 61025: Fault tree analysis (fta),” *IEC Standards Online*, 2006. [Cited on page 17]
- [15] N. G. Leveson, “A systems-theoretic approach to safety in software-intensive systems,” *IEEE Transactions on Dependable and Secure computing*, vol. 1, no. 1, pp. 66–86, 2004. [Cited on page 18]
- [16] A. Abdulkhaleq and S. Wagner, “Experiences with applying stpa to software-intensive systems in the automotive domain,” *STPA Application Areas*, pp. 1–17, 2013. [Cited on page 18]
- [17] A. Abdulkhaleq, D. Lammering, S. Wagner, J. Röder, N. Balbierer, L. Ramsauer, T. Raste, and H. Boehmert, “A systematic approach based on stpa for developing a dependable architecture for fully automated driving vehicles,” *Procedia Engineering*, vol. 179, pp. 41–51, 2017. [Cited on page 18]
- [18] H. L. V. de Matos, A. M. da Cunha, and L. A. V. Dias, “Using design patterns for safety assessment of integrated modular avionics,” in *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*, pp. 4D1–1, IEEE, 2014. [Cited on page 20]
- [19] A. Armoush, *Design patterns for safety-critical embedded systems*. PhD thesis, RWTH Aachen University, 2010. [Cited on page 20]
- [20] J. B. Almeida, M. J. Frade, J. S. Pinto, and S. M. de Sousa, *Rigorous software development: an introduction to program verification*. Springer Science & Business Media, 2011. [Cited on page 22]

-
- [21] F. Khan, S. R. Jan, M. Tahir, S. Khan, and F. Ullah, “Survey: dealing non-functional requirements at architecture level,” *VFAST Transactions on Software Engineering*, vol. 9, no. 2, pp. 7–13, 2016. [Cited on page 22]
- [22] R. E. Bryant, “Graph-based algorithms for boolean function manipulation,” *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986. [Cited on pages 27 and 31]
- [23] A. Mishchenko, “An introduction to zero-suppressed binary decision diagrams,” in *Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, vol. 8, pp. 1–15, Citeseer, 2001. [Cited on page 31]
- [24] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, “Formal methods: Practice and experience,” *ACM computing surveys (CSUR)*, vol. 41, no. 4, p. 19, 2009. [Cited on page 35]
- [25] R. Allen and D. Garlan, *Towards formalized software architectures*. Carnegie-Mellon University. Department of Computer Science, 1992. [Cited on page 35]
- [26] A. Van Lamsweerde, “From system goals to software architecture,” in *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pp. 25–43, Springer, 2003. [Cited on page 35]
- [27] C. A. Hoare, “Proof of correctness of data representations,” *Acta Inf.*, vol. 1, pp. 271–281, Dec. 1972. [Cited on page 35]
- [28] C. B. Jones, *Systematic software development using VDM*, vol. 2. Prentice Hall Englewood Cliffs, 1990. [Cited on page 35]
- [29] E. W. Dijkstra, “Guarded commands, nondeterminacy and formal derivation of programs,” *Commun. ACM*, vol. 18, pp. 453–457, Aug. 1975. [Cited on page 35]
- [30] J. M. Morris, “A theoretical basis for stepwise refinement and the programming calculus,” *Science of Computer programming*, vol. 9, no. 3, pp. 287–306, 1987. [Cited on page 35]

-
- [31] C. Morgan, “The specification statement,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 10, no. 3, pp. 403–419, 1988. [Cited on page 35]
- [32] R. J. R. Back and J. von Wright, “Refinement calculus, part i: Sequential nondeterministic programs,” in *Proceedings on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, REX workshop, (New York, NY, USA), pp. 42–66, Springer-Verlag New York, Inc., 1990. [Cited on page 35]
- [33] R. W. Floyd, “Assigning meanings to programs,” *Proceedings of Symposium on Applied Mathematics*, vol. 19, pp. 19–32, 1967. [Cited on page 35]
- [34] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Commun. ACM*, vol. 12, pp. 576–580, Oct. 1969. [Cited on page 35]
- [35] E. Younger, Z. Luo, K. H. Bennett, and T. Bull, “Reverse engineering concurrent programs using formal modelling and analysis,” in *Proceedings of WCRE’96: 4rd Working Conference on Reverse Engineering*, pp. 239–248, IEEE, 1996. [Cited on page 36]
- [36] M. P. Ward and K. H. Bennett, “Formal methods to aid the evolution of software,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 5, no. 01, pp. 25–47, 1995. [Cited on page 36]
- [37] H. Gomaa, “Software design methods for the design of large-scale real-time systems,” *J. Syst. Softw.*, vol. 25, no. 2, pp. 127–146, 1994. [Cited on page 36]
- [38] A. S. Staines, “A comparison of software analysis and design methods for real time systems,” *World Academy of Science, Engineering and Technology*, pp. 55–59, 2007. [Cited on page 36]
- [39] N. Esfahani, S.-H. Mirian-Hosseiniabadi, and K. Rafati, “Real-time analysis process patterns,” in *Computer Society of Iran Computer Conference*, pp. 777–781, Springer, 2008. [Cited on page 36]
- [40] V. M. Monthe, L. Nana, G. E. Kouamou, and C. Tangha, “A decision support framework for the choice of languages and methods for the design

- of real time embedded systems,” *Journal of Software Engineering and Applications*, vol. 9, no. 07, p. 353, 2016. [Cited on page 36]
- [41] M. Ponnappalli and P. B. Rao, “A comparative study of software architectures for embedded mission critical applications,” in *2016 IEEE 6th International Conference on Advanced Computing (IACC)*, pp. 741–746, IEEE, 2016. [Cited on page 36]
- [42] I. ISO, “Iec/ieee systems and software engineering: Architecture description,” 2011. [Cited on page 38]
- [43] S. Björnander, “Architecture description languages,” *Mrtc. Mdh. Se*, 2011. [Cited on page 38]
- [44] P. C. Clements, “A survey of architecture description languages,” in *Proceedings of the 8th international workshop on software specification and design*, p. 16, IEEE Computer Society, 1996. [Cited on page 38]
- [45] S. Hussain, “Investigating architecture description languages (adls) a systematic literature review,” 2013. [Cited on page 38]
- [46] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, “What industry needs from architectural languages: A survey,” *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 869–891, 2012. [Cited on page 38]
- [47] N. Medvidovic and R. N. Taylor, “A framework for classifying and comparing architecture description languages,” in *Software Engineering ES-EC/FSE’97*, pp. 60–76, Springer, 1997. [Cited on page 38]
- [48] P. Mishra and N. Dutt, “Architecture description languages,” in *Customizable Embedded Processors*, pp. 59–76, Elsevier, 2007. [Cited on page 38]
- [49] I. Sommerville, “Software engineering 9th edition,” *ISBN-10137035152*, 2011. [Cited on page 41]
- [50] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen, “Symbolic model checking,” in *International conference on computer aided verification*, pp. 419–422, Springer, 1996. [Cited on page 49]

-
- [51] “World wide web virtual library on formal methods.” https://formalmethods.wikia.org/wiki/Formal_methods. Accessed: 2021-07-09. [Cited on page 49]
- [52] V. Kumar, “Algorithms for constraint-satisfaction problems: A survey,” *AI magazine*, vol. 13, no. 1, pp. 32–32, 1992. [Cited on page 52]
- [53] A. Petcu and B. Faltings, “Dpop: A scalable method for multiagent constraint optimization,” in *IJCAI 05*, pp. 266–271, 01 2005. [Cited on page 52]
- [54] T. Saxena, *A generic framework for design space exploration*. Vanderbilt University, 2012. [Cited on page 52]
- [55] A. H. Land and A. G. Doig, “An automatic method for solving discrete programming problems,” in *50 Years of Integer Programming 1958-2008*, pp. 105–132, Springer, 2010. [Cited on pages 52, 54, and 61]
- [56] J. Jonsson and K. G. Shin, “A parametrized branch-and-bound strategy for scheduling precedence-constrained tasks on a multiprocessor system,” in *Proceedings of the 1997 International Conference on Parallel Processing (Cat. No. 97TB100162)*, pp. 158–165, IEEE, 1997. [Cited on page 53]
- [57] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998. [Cited on page 53]
- [58] S. Prakash and A. C. Parker, “Synthesis of application-specific multiprocessor architectures,” in *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pp. 8–13, 1991. [Cited on page 53]
- [59] R. Niemann and P. Marwedel, “An algorithm for hardware/software partitioning using mixed integer linear programming,” *Design Automation for Embedded Systems*, vol. 2, no. 2, pp. 165–193, 1997. [Cited on page 53]
- [60] H. Cambazard, D. Mehta, B. O’Sullivan, L. Quesada, M. Ruffini, D. Payne, and L. Doyle, “A combinatorial optimisation approach to the design of dual parented long-reach passive optical networks,” in *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, pp. 785–792, IEEE, 2011. [Cited on page 53]

- [61] H. Kooti, E. Bozorgzadeh, S. Liao, and L. Bao, “Transition-aware real-time task scheduling for reconfigurable embedded systems,” in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pp. 232–237, IEEE, 2010. [Cited on page 53]
- [62] D. Kirov, P. Nuzzo, R. Passerone, and A. Sangiovanni-Vincentelli, “Archex: An extensible framework for the exploration of cyber-physical system architectures,” in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2017. [Cited on pages 53, 89, 90, and 264]
- [63] D. Kirov, P. Nuzzo, R. Passerone, and A. Sangiovanni-Vincentelli, “Optimized selection of wireless network topologies and components via efficient pruning of feasible paths,” in *Proceedings of the 55th Annual Design Automation Conference*, pp. 1–6, 2018. [Cited on page 53]
- [64] J. C. Nash, “The (dantzig) simplex method for linear programming,” *Computing in Science & Engineering*, vol. 2, no. 1, pp. 29–31, 2000. [Cited on pages 53 and 61]
- [65] N. Karmarkar, “A new polynomial-time algorithm for linear programming,” in *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pp. 302–311, 1984. [Cited on page 53]
- [66] J. P. Walser, *Domain-independent local search for linear integer optimization*. PhD thesis, Universität des Saarlandes, 1998. [Cited on pages 54 and 61]
- [67] H. Ramalhinho-Lourenço, O. C. Martin, and T. Stützle, “Iterated local search,” in *Handbook of Metaheuristics* (F. Glover and G. Kochenberger, eds.), pp. 2321–353, Kluwer Academic, 2000. [Cited on page 54]
- [68] T. A. Feo and M. G. Resende, “Greedy randomized adaptive search procedures,” *Journal of global optimization*, vol. 6, no. 2, pp. 109–133, 1995. [Cited on page 54]
- [69] P. J. Van Laarhoven and E. H. Aarts, “Simulated annealing,” in *Simulated annealing: Theory and applications*, pp. 7–15, Springer, 1987. [Cited on page 54]

- [70] F. Glover, *Tabu search fundamentals and uses*. Graduate School of Business, University of Colorado Boulder, 1995. [Cited on pages 54 and 55]
- [71] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by simulated annealing,” *science*, vol. 220, no. 4598, pp. 671–680, 1983. [Cited on pages 54 and 58]
- [72] S. Gupta and L. Bic, “Distributed adaptive simulated annealing for synthesis design space exploration,” tech. rep., University of California, Irvine, CA 92697-3425, 1999. [Cited on page 55]
- [73] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, “System level hardware/software partitioning based on simulated annealing and tabu search,” *Design automation for embedded systems*, vol. 2, no. 1, pp. 5–32, 1997. [Cited on page 55]
- [74] K. Miettinen, *Nonlinear multiobjective optimization*, vol. 12. Springer Science & Business Media, 2012. [Cited on page 56]
- [75] J. Hall and Q. Huangfu, “A high performance dual revised simplex solver,” in *International Conference on Parallel Processing and Applied Mathematics*, pp. 143–151, Springer, 2011. [Cited on page 58]
- [76] P. Kall, S. W. Wallace, and P. Kall, *Stochastic programming*. Springer, 1994. [Cited on page 58]
- [77] D.-T. Peng, K. G. Shin, and T. F. Abdelzaher, “Assignment and scheduling communicating periodic tasks in distributed real-time systems,” *IEEE Transactions on Software Engineering*, vol. 23, no. 12, pp. 745–758, 1997. [Cited on page 58]
- [78] S. Fazlollahi, P. Mandel, G. Becker, and F. Maréchal, “Methods for multi-objective investment and operating optimization of complex energy systems,” *Energy*, vol. 45, no. 1, pp. 12–22, 2012. [Cited on page 58]
- [79] M. Lukasiewicz, M. Glaß, C. Haubelt, and J. Teich, “Efficient symbolic multi-objective design space exploration,” in *2008 Asia and South Pacific Design Automation Conference*, pp. 691–696, IEEE, 2008. [Cited on page 58]
- [80] I. H. Osman and G. Laporte, “Metaheuristics: A bibliography,” *Annals of Operations Research*, no. 63, pp. 513–623, 1996. [Cited on page 58]

-
- [81] S. Greco, J. Figueira, and M. Ehrgott, *Multiple criteria decision analysis*. Springer, 2016. [Cited on page 58]
- [82] F. Glover, “Future paths for integer programming and links to artificial intelligence,” *Computers operations research*, vol. 13, no. 5, pp. 533–549, 1986. [Cited on page 58]
- [83] K. Nonobe and T. Ibaraki, “A tabu search approach to the constraint satisfaction problem as a general problem solver,” *European journal of operational research*, vol. 106, no. 2-3, pp. 599–623, 1998. [Cited on page 58]
- [84] T. Bäck and H.-P. Schwefel, “An overview of evolutionary algorithms for parameter optimization,” *Evolutionary computation*, vol. 1, no. 1, pp. 1–23, 1993. [Cited on pages 58 and 61]
- [85] C. M. Fonseca and P. J. Fleming, “An overview of evolutionary algorithms in multiobjective optimization,” *Evolutionary computation*, vol. 3, no. 1, pp. 1–16, 1995. [Cited on page 58]
- [86] D. E. Goldberg and J. H. Holland, “Genetic algorithms and machine learning,” *Machine Learning*, vol. 3, pp. 95–99, 1988. [Cited on pages 58 and 61]
- [87] J. Kennedy and R. Eberhart, “Particle swarm optimization,” in *Proceedings of ICNN’95-International Conference on Neural Networks*, vol. 4, pp. 1942–1948, IEEE, 1995. [Cited on page 59]
- [88] M. Dorigo, V. Maniezzo, and A. Coloni, “Ant system: optimization by a colony of cooperating agents,” *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 1, pp. 29–41, 1996. [Cited on page 59]
- [89] M. Gelfond and V. Lifschitz, “The stable model semantics for logic programming,” in *ICLP/SLP*, vol. 88, pp. 1070–1080, 1988. [Cited on page 59]
- [90] T. Crick, *Superoptimisation: provably optimal code generation using answer set programming*. PhD thesis, University of Bath, 2009. [Cited on page 59]

-
- [91] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*, vol. 185. IOS press, 2009. [Cited on page 59]
- [92] C. Barrett and C. Tinelli, “Satisfiability modulo theories,” in *Handbook of Model Checking*, pp. 305–343, Springer, 2018. [Cited on page 59]
- [93] A. Cimatti, A. Griggio, and R. Sebastiani, “Computing small unsatisfiable cores in satisfiability modulo theories,” *Journal of Artificial Intelligence Research*, vol. 40, pp. 701–728, 2011. [Cited on page 59]
- [94] R. Nieuwenhuis and A. Oliveras, “On sat modulo theories and optimization problems,” in *International conference on theory and applications of satisfiability testing*, pp. 156–169, Springer, 2006. [Cited on page 60]
- [95] A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani, and C. Stenico, “Satisfiability modulo the theory of costs: Foundations and applications,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 99–113, Springer, 2010. [Cited on page 60]
- [96] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik, “Symbolic optimization with smt solvers,” *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 607–618, 2014. [Cited on page 60]
- [97] N. Bjørner, A.-D. Phan, and L. Fleckenstein, “vz-an optimizing smt solver,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 194–199, Springer, 2015. [Cited on page 60]
- [98] R. Sebastiani and P. Trentin, “Optimathsat: A tool for optimization modulo theories,” *Journal of Automated Reasoning*, vol. 64, no. 3, pp. 423–460, 2020. [Cited on pages 60 and 98]
- [99] C. Bazgan, S. Ruzika, C. Thielen, and D. Vanderpooten, “The power of the weighted sum scalarization for approximating multiobjective optimization problems,” *arXiv preprint arXiv:1908.01181*, 2019. [Cited on page 61]
- [100] A. Charnes and W. W. Cooper, “Management models and industrial applications of linear programming,” *Management science*, vol. 4, no. 1, pp. 38–91, 1957. [Cited on page 61]

-
- [101] E. B. Baum, “Towards practical ‘neural’ computation for combinatorial optimization problems,” in *AIP Conference Proceedings*, vol. 151, pp. 53–58, American Institute of Physics, 1986. [Cited on page 61]
- [102] T. Givargis, F. Vahid, and J. Henkel, “System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip,” in *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No. 01CH37281)*, pp. 25–30, IEEE, 2001. [Cited on page 61]
- [103] K. Neubauer, P. Wanko, T. Schaub, and C. Haubelt, “Exact multi-objective design space exploration using aspmt,” in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 257–260, IEEE, 2018. [Cited on page 61]
- [104] C. Lo and P. Chow, “Hierarchical modelling of generators in design-space exploration,” in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 186–194, IEEE, 2020. [Cited on page 61]
- [105] T. Schlichter, C. Haubelt, and J. Teich, “Improving ea-based design space exploration by utilizing symbolic feasibility tests,” in *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pp. 1945–1952, 2005. [Cited on page 62]
- [106] P. Traverse, I. Lacaze, and J. Souyris, “A process toward total dependability-airbus fly-by-wire paradigm.,” in *EDCC*, p. 1, Springer, 2005. [Cited on page 63]
- [107] R. John, “Partitioning in avionics architectures: requirements, mechanisms, and assurance,” *NASA Contractor Report*, 1999. [Cited on page 63]
- [108] P. Sinha, “Architectural design and reliability analysis of a fail-operational brake-by-wire system from iso 26262 perspectives,” *Reliability Engineering & System Safety*, vol. 96, no. 10, pp. 1349–1359, 2011. [Cited on page 63]
- [109] J. Wei, J. M. Snider, J. Kim, J. M. Dolan, R. Rajkumar, and B. Litkouhi, “Towards a viable autonomous driving research platform,” in *2013 IEEE*

- Intelligent Vehicles Symposium (IV)*, pp. 763–770, IEEE, 2013. [Cited on page 64]
- [110] M. Buechel, J. Frtunikj, K. Becker, S. Sommer, C. Buckl, M. Armbruster, A. Marek, A. Zirkler, C. Klein, and A. Knoll, “An automated electric vehicle prototype showing new trends in automotive architectures,” in *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, pp. 1274–1279, IEEE, 2015. [Cited on page 64]
- [111] M. Güdemann and F. Ortmeier, “Model-based multi-objective safety optimization,” in *International Conference on Computer Safety, Reliability, and Security*, pp. 423–436, Springer, 2011. [Cited on page 64]
- [112] S. Lazarova-Molnar, H. R. Shaker, and N. Mohamed, “Reliability of cyber physical systems with focus on building management systems,” in *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*, pp. 1–6, IEEE, 2016. [Cited on page 64]
- [113] W. Nace and P. Koopman, “A product family approach to graceful degradation,” in *IFIP Working Conference on Distributed and Parallel Embedded Systems*, pp. 131–140, Springer, 2000. [Cited on page 64]
- [114] C. P. Shelton, P. Koopman, and W. Nace, “A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems,” in *8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003), 15-17 January 2003, Guadalajara, Mexico*, pp. 156–163, IEEE Computer Society, 2003. [Cited on page 64]
- [115] C. P. Shelton, P. Koopman, and W. Nace, “A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems,” in *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems, 2003.(WORDS 2003).*, pp. 156–163, IEEE, 2003. [Cited on page 65]
- [116] C. Shelton and P. Koopman, “Using architectural properties to model and measure graceful degradation,” in *Architecting dependable systems*, pp. 267–289, Springer, 2003. [Cited on page 65]

-
- [117] C. P. Shelton and P. Koopman, “Improving system dependability with functional alternatives,” in *International Conference on Dependable Systems and Networks, 2004*, pp. 295–304, IEEE, 2004. [Cited on page 65]
- [118] P. Emberson and I. Bate, “Extending a task allocation algorithm for graceful degradation of real-time distributed embedded systems,” in *2008 Real-Time Systems Symposium*, pp. 270–279, IEEE, 2008. [Cited on page 65]
- [119] P. Emberson, *Searching for flexible solutions to task allocation problems*. PhD thesis, University of York, 2009. [Cited on page 65]
- [120] M. Trapp, R. Adler, M. Förster, and J. Junger, “Runtime adaptation in safety-critical automotive systems,” *Software Engineering*, pp. 1–8, 2007. [Cited on page 65]
- [121] M. Glaß, M. Lukasiewicz, C. Haubelt, and J. Teich, “Incorporating graceful degradation into embedded system design,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 320–323, European Design and Automation Association, 2009. [Cited on page 66]
- [122] D. Penha, G. Weiss, and A. Stante, “Pattern-based approach for designing fail-operational safety-critical embedded systems,” in *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*, pp. 52–59, IEEE, 2015. [Cited on page 66]
- [123] J. Kim, G. Bhatia, R. Rajkumar, and M. Jochim, “Safer: System-level architecture for failure evasion in real-time applications,” in *2012 IEEE 33rd Real-Time Systems Symposium*, pp. 227–236, IEEE, 2012. [Cited on page 67]
- [124] J. Kim, R. R. Rajkumar, and M. Jochim, “Towards dependable autonomous driving vehicles: a system-level approach,” *ACM SIGBED Review*, vol. 10, no. 1, pp. 29–32, 2013. [Cited on page 67]
- [125] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri, “Safety, dependability and performance analysis of extended aadl models,” *The Computer Journal*, vol. 54, no. 5, pp. 754–775, 2010. [Cited on page 67]

-
- [126] K. Becker, *Software Deployment Analysis for Mixed Reliability Automotive Systems*. PhD thesis, Technische Universität München, 2017. [Cited on page 67]
- [127] A. Hamann, *Iterative design space exploration and robustness optimization for embedded systems*. Cuvillier Verlag, 2008. [Cited on page 68]
- [128] L. Grunske, “Identifying good architectural design alternatives with multi-objective optimization strategies,” in *Proceedings of the 28th international conference on Software engineering*, pp. 849–852, ACM, 2006. [Cited on pages 68, 77, and 78]
- [129] M. Mikic-Rakic, S. Malek, N. Beckman, and N. Medvidovic, “A tailorable environment for assessing the quality of deployment architectures in highly distributed settings,” in *International Working Conference on Component Deployment*, pp. 1–17, Springer, 2004. [Cited on page 68]
- [130] M. Junker, *Specification and Analysis of Availability for Software-Intensive Systems*. PhD thesis, Technische Universität München, 2016. [Cited on page 69]
- [131] M. Broy and K. Stølen, *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer Science & Business Media, 2012. [Cited on page 69]
- [132] P. K. Saraswat, P. Pop, and J. Madsen, “Task migration for fault-tolerance in mixed-criticality embedded systems,” *ACM SIGBED Review*, vol. 6, no. 3, p. 6, 2009. [Cited on page 69]
- [133] S. Baruah, H. Li, and L. Stougie, “Towards the design of certifiable mixed-criticality systems,” in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 13–22, IEEE, 2010. [Cited on page 69]
- [134] S. Voss and B. Schätz, “Deployment and scheduling synthesis for mixed-critical shared-memory applications,” in *2013 20th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*, pp. 100–109, IEEE, 2013. [Cited on page 70]

-
- [135] A. Thekkilakattil, R. Dobrin, and S. Punnekkat, “Mixed criticality scheduling in fault-tolerant distributed real-time systems,” in *2014 International Conference on Embedded Systems (ICES)*, pp. 92–97, IEEE, 2014. [Cited on page 70]
- [136] D. Tămaş-Selicean, P. Pop, and J. Madsen, “Design of mixed-criticality applications on distributed real-time systems,” *Technical University of Denmark*, 2014. [Cited on page 70]
- [137] G. Xie, G. Zeng, L. Liu, R. Li, and K. Li, “High performance real-time scheduling of multiple mixed-criticality functions in heterogeneous distributed embedded systems,” *Journal of Systems Architecture*, vol. 70, pp. 3–14, 2016. [Cited on page 71]
- [138] A. Cansado, C. Canal, G. Salaün, and J. Cubo, “A formal framework for structural reconfiguration of components under behavioural adaptation,” *Electronic Notes in Theoretical Computer Science*, vol. 263, pp. 95–110, 2010. [Cited on page 71]
- [139] B. Becker, H. Giese, S. Neumann, M. Schenck, and A. Treffer, “Model-based extension of autosar for architectural online reconfiguration,” in *International Conference on Model Driven Engineering Languages and Systems*, pp. 83–97, Springer, 2009. [Cited on page 71]
- [140] P. Oreizy, N. Medvidovic, and R. N. Taylor, “Architecture-based runtime software evolution,” in *Proceedings of the 20th international conference on Software engineering*, pp. 177–186, IEEE, 1998. [Cited on page 72]
- [141] N. Medvidovic and R. N. Taylor, “A classification and comparison framework for software architecture description languages,” *IEEE Transactions on software engineering*, vol. 26, no. 1, pp. 70–93, 2000. [Cited on page 72]
- [142] P. Oreizy, N. Medvidovic, and R. N. Taylor, “Runtime software adaptation: framework, approaches, and styles,” in *Companion of the 30th international conference on Software engineering*, pp. 899–910, ACM, 2008. [Cited on page 72]
- [143] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger, “A survey of self-management in dynamic software architecture specifications,” in

- Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pp. 28–33, ACM, 2004. [Cited on page 72]
- [144] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, “Dynamic software product lines,” *Computer*, vol. 41, no. 4, pp. 93–95, 2008. [Cited on page 72]
- [145] T. Dinkelaker, R. Mitschke, K. Fetzer, and M. Mezini, “A dynamic software product line approach using aspect models at runtime,” in *Proceedings of the 1st Workshop on Composition and Variability*, pp. 180–220, CEUR Workshop, 2010. [Cited on page 72]
- [146] J. Simmonds and M. C. Bastarrica, “Modeling variability in software process lines,” *Departamento de Ciencias de la Computación. Universidad de Chile*, 2011. [Cited on page 72]
- [147] N. Loughran, P. Sánchez, A. Garcia, and L. Fuentes, “Language support for managing variability in architectural models,” in *International Conference on Software Composition*, pp. 36–51, Springer, 2008. [Cited on page 72]
- [148] C. Cetina, V. Pelechano, P. Trinidad, and A. R. Cortés, “An architectural discussion on dspl,” in *SPLC (2)*, pp. 59–68, Citeseer, 2008. [Cited on page 72]
- [149] M. Salehie and L. Tahvildari, “Self-adaptive software: Landscape and research challenges,” *ACM transactions on autonomous and adaptive systems (TAAS)*, vol. 4, no. 2, p. 14, 2009. [Cited on page 72]
- [150] G. D. Rodosek, K. Geihs, H. Schmeck, and B. Stiller, “Self-healing systems: Foundations and challenges,” *Self-Healing and Self-Adaptive Systems*, no. 09201, 2009. [Cited on page 72]
- [151] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” *Computer*, no. 1, pp. 41–50, 2003. [Cited on page 72]
- [152] G. Mühl, M. Werner, M. A. Jaeger, K. Herrmann, and H. Parzyjegl, “On the definitions of self-managing and self-organizing systems,” in *Communication in Distributed Systems-15. ITG/GI Symposium*, pp. 1–11, VDE, 2007. [Cited on page 72]

-
- [153] S. Stein, M. Neukirchner, and R. Ernst, “Admission control and self-configuration in the epc framework,” in *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pp. 364–371, IEEE, 2011. [Cited on page 72]
- [154] M. Bozzano, A. Cimatti, and C. Mattarei, “Formal reliability analysis of redundancy architectures,” *Formal Aspects of Computing*, vol. 31, no. 1, pp. 59–94, 2019. [Cited on pages 73, 111, 117, 126, 129, 130, 131, 134, 135, 160, 163, and 191]
- [155] K. L. McMillan, “The smv language,” *Cadence Berkeley Labs*, pp. 1–49, 1999. [Cited on pages 73, 112, and 135]
- [156] B. Bittner, M. Bozzano, R. Cavada, A. Cimatti, M. Gario, A. Griggio, C. Mattarei, A. Micheli, and G. Zampedri, “The xsap safety analysis platform,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 533–539, Springer, 2016. [Cited on pages 73, 74, 112, and 135]
- [157] S. Lee, J.-i. Jung, and I. Lee, “Voting structures for cascaded triple modular redundant modules,” *IEICE Electronics Express*, vol. 4, no. 21, pp. 657–664, 2007. [Cited on page 73]
- [158] J.-P. Katoen, M. Khattri, and I. Zapreevt, “A markov reward model checker,” in *Second International Conference on the Quantitative Evaluation of Systems (QEST’05)*, pp. 243–244, IEEE, 2005. [Cited on page 73]
- [159] M. Kwiatkowska, G. Norman, and D. Parker, “Prism: Probabilistic model checking for performance and reliability analysis,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 40–45, 2009. [Cited on page 73]
- [160] K. S. Trivedi, “Sharpe 2002: Symbolic hierarchical automated reliability and performance evaluator,” in *Proceedings International Conference on Dependable Systems and Networks*, p. 544, IEEE, 2002. [Cited on page 73]
- [161] M. Kwiatkowska, G. Norman, and D. Parker, “Prism 4.0: Verification of probabilistic real-time systems,” in *International conference on computer aided verification*, pp. 585–591, Springer, 2011. [Cited on page 73]

-
- [162] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen, “Uppaal smc tutorial,” *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 4, pp. 397–415, 2015. [Cited on page 73]
- [163] T. Lanfang, T. Qingping, and L. Jianli, “Specification and verification of the triple-modular redundancy fault tolerant system using csp,” in *DEPEND 2011, The Fourth International Conference on Dependability*, pp. 14–17, 2011. [Cited on page 73]
- [164] A. Hartmanns, “Modest-a unified language for quantitative models,” in *Proceeding of the 2012 Forum on Specification and Design Languages*, pp. 44–51, IEEE, 2012. [Cited on page 74]
- [165] O. Lisagor, T. Kelly, and R. Niu, “Model-based safety assessment: Review of the discipline and its challenges,” in *The Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety*, pp. 625–632, IEEE, 2011. [Cited on page 74]
- [166] J. Delange and P. Feiler, “Architecture fault modeling with the aadl error-model annex,” in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 361–368, IEEE, 2014. [Cited on page 74]
- [167] S. Kabir, Y. Papadopoulos, M. Walker, D. Parker, J. I. Aizpurua, J. Lampe, and E. Rūde, “A model-based extension to hip-hops for dynamic fault propagation studies,” in *International Symposium on Model-Based Safety and Assessment*, pp. 163–178, Springer, 2017. [Cited on page 74]
- [168] A. Arnold, G. Point, A. Griffault, and A. Rauzy, “The altarica formalism for describing concurrent systems,” *Fundamenta Informaticae*, vol. 40, no. 2, 3, pp. 109–124, 1999. [Cited on page 74]
- [169] M. Batteux, T. Prosvirnova, and A. Rauzy, “System structure modeling language (s2ml).” working paper or preprint, Dec. 2015. [Cited on page 74]
- [170] P. Nuzzo, N. Bajaj, M. Masin, D. Kirov, R. Passerone, and A. L. Sangiovanni-Vincentelli, “Optimized selection of reliable and cost-effective safety-critical system architectures,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2109–2123, 2019. [Cited on pages 74 and 264]

-
- [171] M. Buyse, R. Delmas, and Y. Hamadi, “Alpacas: A language for parametric assessment of critical architecture safety,” in *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021. [Cited on page 74]
- [172] M. Chaudron, S. Larsson, and I. Crnkovic, “Component-based development process and component lifecycle,” *Journal of Computing and Information Technology*, vol. 13, no. 4, pp. 321–327, 2005. [Cited on page 76]
- [173] A. Sangiovanni-Vincentelli and G. Martin, “Platform-based design and software design methodology for embedded systems,” *IEEE Design & Test of Computers*, vol. 18, no. 6, pp. 23–33, 2001. [Cited on page 76]
- [174] G. Ascia, V. Catania, and M. Palesi, “An evolutionary approach for pareto-optimal configurations in soc platforms,” in *SOC Design Methodologies*, pp. 157–168, Springer, 2002. [Cited on pages 76 and 78]
- [175] S. Neema, J. Sztipanovits, G. Karsai, and K. Butts, “Constraint-based design-space exploration and model synthesis,” in *International Workshop on Embedded Software*, pp. 290–305, Springer, 2003. [Cited on pages 77 and 78]
- [176] H. Neema, Z. Lattmann, P. Meijer, J. Klingler, S. Neema, T. Bapty, J. Sztipanovits, and G. Karsai, “Design space exploration and manipulation for cyber physical systems,” in *IFIP First International Workshop on Design Space Exploration of Cyber-Physical Systems (IDEAL?2014)*, Springer-Verlag Berlin Heidelberg, p. 8, 2014. [Cited on pages 77 and 78]
- [177] P. Manolios and V. Papavasileiou, “Virtual integration of cyber-physical systems by verification,” in *Proc. AVICPS*, p. 65, Citeseer, 2010. [Cited on pages 77 and 78]
- [178] D. E. Fyffe, W. W. Hines, and N. K. Lee, “System reliability allocation and a computational algorithm,” *IEEE Transactions on Reliability*, vol. 17, no. 2, pp. 64–69, 1968. [Cited on pages 78, 79, and 249]
- [179] Y. Nakagawa and S. Miyazaki, “Surrogate constraints algorithm for reliability optimization problems with two constraints,” *IEEE Transactions on Reliability*, vol. 30, no. 2, pp. 175–180, 1981. [Cited on pages 78, 79, and 250]

- [180] M. Soto, A. Rossi, and M. Sevaux, “Two iterative metaheuristic approaches to dynamic memory allocation for embedded systems,” in *European Conference on Evolutionary Computation in Combinatorial Optimization*, pp. 250–261, Springer, 2011. [Cited on page 78]
- [181] P. Van Huong and N. N. Binh, “Embedded system architecture design and optimization at the model level,” *International Journal of Computer and Communication Engineering*, vol. 1, no. 4, p. 345, 2012. [Cited on page 78]
- [182] K. Settaluri, A. Haj-Ali, Q. Huang, K. Hakhamaneshi, and B. Nikolic, “Autockt: Deep reinforcement learning of analog circuit designs,” in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 490–495, IEEE, 2020. [Cited on page 78]
- [183] P. Terway, K. Hamidouche, and N. K. Jha, “Dispatch: Design space exploration of cyber-physical systems,” *arXiv preprint arXiv:2009.10214*, 2020. [Cited on page 78]
- [184] D. W. Coit and A. E. Smith, “Solving the redundancy allocation problem using a combined neural network/genetic algorithm approach,” *Computers & operations research*, vol. 23, no. 6, pp. 515–526, 1996. [Cited on pages 80 and 81]
- [185] S. Kulturel-Konak, A. E. Smith, and D. W. Coit, “Efficiently solving the redundancy allocation problem using tabu search,” *IIE transactions*, vol. 35, no. 6, pp. 515–526, 2003. [Cited on pages 80, 81, and 250]
- [186] Y.-C. Liang and A. E. Smith, “An ant colony optimization algorithm for the redundancy allocation problem (rap),” *IEEE Transactions on reliability*, vol. 53, no. 3, pp. 417–423, 2004. [Cited on pages 80 and 81]
- [187] A. Jhumka, S. Klaus, and S. A. Huss, “A dependability-driven system-level design approach for embedded systems,” in *Design, Automation and Test in Europe*, pp. 372–377, IEEE, 2005. [Cited on pages 80 and 81]
- [188] M. Glaß, M. Lukasiewicz, T. Streichert, C. Haubelt, and J. Teich, “Interactive presentation: Reliability-aware system synthesis,” in *Proceedings of the conference on Design, automation and test in Europe*, pp. 409–414, EDA Consortium, 2007. [Cited on pages 80 and 81]

-
- [189] T. Streichert, M. Glaß, C. Haubelt, and J. Teich, “Design space exploration of reliable networked embedded systems,” *Journal of Systems Architecture*, vol. 53, no. 10, pp. 751–763, 2007. [Cited on pages 80 and 81]
- [190] F. Reimann, M. Glaß, M. Lukasiewicz, J. Keinert, C. Haubelt, and J. Teich, “Symbolic voter placement for dependability-aware system synthesis,” in *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pp. 237–242, ACM, 2008. [Cited on pages 80 and 81]
- [191] V. Izosimov, I. Polian, P. Pop, P. Eles, and Z. Peng, “Analysis and optimization of fault-tolerant embedded systems with hardened processors,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 682–687, European Design and Automation Association, 2009. [Cited on pages 80 and 81]
- [192] I. Meedeniya, A. Aleti, and B. Buhnova, “Redundancy allocation in automotive systems using multi-objective optimisation,” in *Symposium of Avionics/Automotive Systems Engineering (SAASE09), San Diego, CA*, 2009. [Cited on pages 80 and 81]
- [193] M. Sheikhalishahi, V. Ebrahimipour, H. Shiri, H. Zaman, and M. Jeehoonian, “A hybrid ga–pso approach for reliability optimization in redundancy allocation problem,” *The International Journal of Advanced Manufacturing Technology*, vol. 68, no. 1-4, pp. 317–338, 2013. [Cited on page 81]
- [194] K. Delmas, R. Delmas, and C. Pagetti, “Automatic architecture hardening using safety patterns,” in *International Conference on Computer Safety, Reliability, and Security*, pp. 283–296, Springer, 2014. [Cited on page 81]
- [195] K. Delmas, R. Delmas, and C. Pagetti, “Smt-based synthesis of fault-tolerant architectures,” in *International Conference on Computer Safety, Reliability, and Security*, pp. 287–302, Springer, 2017. [Cited on page 81]
- [196] M. A. Ardakan and M. T. Rezvan, “Multi-objective optimization of reliability–redundancy allocation problem with cold-standby strategy using nsga-ii,” *Reliability Engineering & System Safety*, vol. 172, pp. 225–238, 2018. [Cited on page 81]

-
- [197] S. Grüner, S. Malakuti, J. Schmitt, T. Terzimehic, M. Wenger, and H. Elfaham, “Alternatives for flexible deployment architectures in industrial automation systems,” in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, pp. 35–42, IEEE, 2018. [Cited on pages 81 and 87]
- [198] T. Terzimehic, S. Voss, and M. Wenger, “Using design space exploration to calculate deployment configurations of iec 61499-based systems,” in *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, pp. 881–886, IEEE, 2018. [Cited on pages 81 and 87]
- [199] A. Zaretalab, V. Hajipour, and M. Tavana, “Redundancy allocation problem with multi-state component systems and reliable supplier selection,” *Reliability Engineering & System Safety*, vol. 193, p. 106629, 2020. [Cited on page 81]
- [200] P. van Stralen and A. D. Pimentel, “A SAFE approach towards early design space exploration of fault-tolerant multimedia mpsoCs,” in *Proceedings of the 10th International Conference on Hardware/Software Code-sign and System Synthesis, CODES+ISSS 2012, part of ESWeek '12 Eighth Embedded Systems Week, Tampere, Finland, October 7-12, 2012* (A. Jerraya, L. P. Carloni, N. Chang, and F. Fummi, eds.), pp. 393–402, ACM, 2012. [Cited on page 81]
- [201] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, “Automated reasoning on feature models,” in *International Conference on Advanced Information Systems Engineering*, pp. 491–503, Springer, 2005. [Cited on pages 81 and 83]
- [202] K. Czarnecki, M. Antkiewicz, C. H. P. Kim, S. Lau, and K. Pietroszek, “Model-driven software product lines,” in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 126–127, 2005. [Cited on page 82]
- [203] J. White, D. C. Schmidt, E. Wuchner, and A. Nechypurenko, “Automating product-line variant selection for mobile devices,” in *11th International Software Product Line Conference (SPLC 2007)*, pp. 129–140, IEEE, 2007. [Cited on pages 82 and 83]

-
- [204] C. Seidl, I. Schaefer, and U. Aßmann, “Deltaecore—a model-based delta language generation framework,” *Modellierung 2014*, 2014. [Cited on pages 82 and 83]
- [205] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, “Delta-oriented programming of software product lines,” in *International Conference on Software Product Lines*, pp. 77–91, Springer, 2010. [Cited on page 82]
- [206] C. Pietsch, T. Kehrer, U. Kelter, D. Reuling, and M. Ohrndorf, “Sipl—a delta-based modeling framework for software product line engineering,” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 852–857, IEEE, 2015. [Cited on pages 82 and 83]
- [207] A. Durán, D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés, “Flame: a formal framework for the automated analysis of software product lines validated by automated specification testing,” *Software & Systems Modeling*, vol. 16, no. 4, pp. 1049–1082, 2017. [Cited on page 83]
- [208] R. Schröter, S. Krieter, T. Thüm, F. Benduhn, and G. Saake, “Compositional analyses of highly-configurable systems with feature-model interfaces,” *Software Engineering 2017*, 2017. [Cited on page 83]
- [209] F. M. Kifetew, D. Muñante, J. Gorroñoigoitia, A. Siena, A. Susi, and A. Perini, “Grammar based genetic programming for software configuration problem,” in *International Symposium on Search Based Software Engineering*, pp. 130–136, Springer, 2017. [Cited on page 83]
- [210] F. Schwägerl and B. Westfechtel, “Integrated revision and variation control for evolving model-driven software product lines,” *Software and Systems Modeling*, vol. 18, no. 6, pp. 3373–3420, 2019. [Cited on page 83]
- [211] Z.-W. Jiang, H.-C. Chen, T.-C. Chen, and Y.-W. Chang, “Challenges and solutions in modern vlsi placement,” in *2007 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pp. 1–5, IEEE, 2007. [Cited on pages 84 and 86]

- [212] R. Saraswat and B. Eames, “Finite domain constraints based delay aware placement tool for fpoas,” in *2008 International Conference on Reconfigurable Computing and FPGAs*, pp. 145–150, IEEE, 2008. [Cited on pages 84 and 86]
- [213] X. Chen, G. Lin, J. Chen, and W. Zhu, “An adaptive hybrid genetic algorithm for vlsi standard cell placement problem,” in *2016 3rd International Conference on Information Science and Control Engineering (ICISCE)*, pp. 163–167, IEEE, 2016. [Cited on pages 84 and 86]
- [214] A. Goldie and A. Mirhoseini, “Placement optimization with deep reinforcement learning,” in *Proceedings of the 2020 International Symposium on Physical Design*, pp. 3–7, 2020. [Cited on page 84]
- [215] R. Saraswat, *A finite domain constraint approach for placement and routing of coarse-grained reconfigurable architectures*. Utah State University, 2010. [Cited on pages 84 and 86]
- [216] J. Chabarek, J. Sommers, P. Barford, C. Estan, D. Tsiang, and S. Wright, “Power awareness in network design and routing,” in *IEEE INFOCOM 2008-The 27th Conference on Computer Communications*, pp. 457–465, IEEE, 2008. [Cited on pages 84 and 86]
- [217] F.-Y. Chang, R.-S. Tsay, W.-K. Mak, and S.-H. Chen, “Mana: A shortest path maze algorithm under separation and minimum length nanometer rules,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 10, pp. 1557–1568, 2013. [Cited on page 85]
- [218] A. B. Kahng, L. Wang, and B. Xu, “Tritonroute: An initial detailed router for advanced vlsi technologies,” in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2018. [Cited on page 85]
- [219] Y. Zhang and C. Chu, “Regularroute: An efficient detailed router with regular routing patterns,” in *Proceedings of the 2011 international symposium on Physical design*, pp. 45–52, 2011. [Cited on page 85]
- [220] G. Liu, W. Zhu, S. Xu, Z. Zhuang, Y.-C. Chen, and G. Chen, “Efficient vlsi routing algorithm employing novel discrete pso and multi-stage

- transformation,” *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–16, 2020. [Cited on page 85]
- [221] K. Neubauer, P. Wanko, T. Schaub, and C. Haubelt, “Exact multi-objective design space exploration using aspmt,” in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 257–260, IEEE, 2018. [Cited on page 85]
- [222] M.-K. Hsu, Y.-F. Chen, C.-C. Huang, S. Chou, T.-H. Lin, T.-C. Chen, and Y.-W. Chang, “Ntuplace4h: A novel routability-driven placement algorithm for hierarchical mixed-size circuit designs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 12, pp. 1914–1927, 2014. [Cited on page 86]
- [223] J. Lu, P. Chen, C.-C. Chang, L. Sha, J. Dennis, H. Huang, C.-C. Teng, and C.-K. Cheng, “eplace: Electrostatics based placement using nesterov’s method,” in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2014. [Cited on page 86]
- [224] C.-K. Cheng, A. B. Kahng, I. Kang, and L. Wang, “Replace: Advancing solution quality and routability validation in global placement,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 9, pp. 1717–1730, 2018. [Cited on page 86]
- [225] Y. Lin, Z. Jiang, J. Gu, W. Li, S. Dhar, H. Ren, B. Khailany, and D. Z. Pan, “Dreamplace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020. [Cited on page 86]
- [226] P. B. Kruchten, “The 4+ 1 view model of architecture,” *IEEE software*, vol. 12, no. 6, pp. 42–50, 1995. [Cited on page 86]
- [227] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little, *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002. [Cited on page 86]
- [228] J. Kramer and J. Magee, “Self-managed systems: an architectural challenge,” in *2007 Future of Software Engineering*, pp. 259–268, IEEE Computer Society, 2007. [Cited on page 86]

- [229] N. Arshad, D. Heimbigner, and A. L. Wolf, “Deployment and dynamic re-configuration planning for distributed software systems,” in *Proceedings. 15th IEEE International Conference on Tools with Artificial Intelligence*, pp. 39–46, IEEE, 2003. [Cited on page 86]
- [230] J. Carlson, J. Feljan, J. Maki-Turja, and M. Sjodin, “Deployment modelling and synthesis in a component model for distributed embedded systems,” in *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 74–82, IEEE, 2010. [Cited on page 86]
- [231] O. Bushehrian, “Automatic object deployment for software performance enhancement,” *IET software*, vol. 5, no. 4, pp. 375–384, 2011. [Cited on page 86]
- [232] J. White, B. Dougherty, C. Thompson, and D. C. Schmidt, “Scatterd: Spatial deployment optimization with hybrid heuristic/evolutionary algorithms,” *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 6, no. 3, p. 18, 2011. [Cited on page 86]
- [233] A. Petricic, “Predictable dynamic deployment of components in embedded systems,” in *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 1128–1129, IEEE, 2011. [Cited on page 86]
- [234] S. Kugele, G. Pucea, R. Popa, L. Dieudonné, and H. Eckardt, “On the deployment problem of embedded systems,” in *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pp. 158–167, IEEE, 2015. [Cited on page 87]
- [235] S. Zverlov, M. Khalil, and M. Chaudhary, “Pareto-efficient deployment synthesis for safety-critical applications in seamless model-based development,” in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, (TOULOUS, France), Jan. 2016. [Cited on page 87]
- [236] E. Ábrahám, F. Corzilius, E. B. Johnsen, G. Kremer, and J. Mauro, “Zephyrus2: On the fly deployment optimization using smt and cp technologies,” in *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, pp. 229–245, Springer, 2016. [Cited on page 87]

- [237] H. Javaid and S. Parameswaran, “A design flow for application specific heterogeneous pipelined multiprocessor systems,” in *Proceedings of the 46th Annual Design Automation Conference*, pp. 250–253, 2009. [Cited on page 89]
- [238] E. L. de Souza Carvalho, N. L. V. Calazans, and F. G. Moraes, “Dynamic task mapping for mpsoCs,” *IEEE Design & Test of Computers*, vol. 27, no. 5, pp. 26–35, 2010. [Cited on page 89]
- [239] A. K. Singh, T. Srikanthan, A. Kumar, and W. Jigang, “Communication-aware heuristics for run-time task mapping on noc-based mpsoC platforms,” *Journal of Systems Architecture*, vol. 56, no. 7, pp. 242–255, 2010. [Cited on page 89]
- [240] J. Huang, A. Raabe, C. Buckl, and A. Knoll, “A workflow for runtime adaptive task allocation on heterogeneous mpsoCs,” in *2011 Design, Automation Test in Europe*, pp. 1–6, 2011. [Cited on page 89]
- [241] L. Chen, T. Marconi, and T. Mitra, “Online scheduling for multi-core shared reconfigurable fabric,” *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 582–585, 2012. [Cited on page 89]
- [242] C. Bolchini, M. Carminati, A. Miele, A. Das, A. Kumar, and B. Veeravalli, “Run-time mapping for reliable many-cores based on energy/performance trade-offs,” in *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, pp. 58–64, IEEE, 2013. [Cited on page 89]
- [243] A. M. Frisch, B. Hnich, I. Miguel, B. M. Smith, and T. Walsh, “Towards csp model reformulation at multiple levels of abstraction,” 2002. [Cited on pages 89 and 90]
- [244] A. Weichslgartner, D. Gangadharan, S. Wildermann, M. Glaß, and J. Teich, “Daarm: Design-time application analysis and run-time mapping for predictable execution in many-core systems,” in *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pp. 1–10, IEEE, 2014. [Cited on pages 89 and 90]
- [245] A. Goens, R. Khasanov, J. Castrillon, M. Hähnel, T. Smejkal, and H. Härtig, “Tetris: a multi-application run-time system for predictable

- execution of static mappings,” in *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*, pp. 11–20, 2017. [Cited on pages 89 and 90]
- [246] C. Erbas, S. Cerav-Erbas, and A. D. Pimentel, “Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design,” *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 3, pp. 358–374, 2006. [Cited on page 90]
- [247] P.-E. Hladik, H. Cambazard, A.-M. Déplanche, and N. Jussien, “Solving a real-time allocation problem with constraint programming,” *Journal of Systems and Software*, vol. 81, no. 1, pp. 132–149, 2008. [Cited on page 90]
- [248] A. N. Letchford, Q. Ni, and Z. Zhong, “An exact algorithm for a resource allocation problem in mobile wireless communications,” *Computational Optimization and Applications*, vol. 68, no. 2, pp. 193–208, 2017. [Cited on page 90]
- [249] J. Madsen and P. Bjorn-Jorgensen, “Embedded system synthesis under memory constraints,” in *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES’99)(IEEE Cat. No. 99TH8450)*, pp. 188–192, IEEE, 1999. [Cited on pages 90 and 91]
- [250] R. Szymanek and K. Kuchcinski, “Design space exploration in system level synthesis under memory constraints,” in *Proceedings 25th EURO-MICRO Conference. Informatics: Theory and Practice for the New Millennium*, vol. 1, pp. 29–36, IEEE, 1999. [Cited on pages 90 and 91]
- [251] C. Le Pape *et al.*, “Constraint-based scheduling: A tutorial,” 2005. [Cited on page 91]
- [252] K. Kuchcinski, “Constraints-driven scheduling and resource assignment,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 8, no. 3, pp. 355–383, 2003. [Cited on page 91]
- [253] J. Porter, G. Karsai, and J. Sztipanovits, “Towards a time-triggered schedule calculation tool to support model-based embedded software design,” in *Proceedings of the seventh ACM international conference on Embedded software*, pp. 167–176, ACM, 2009. [Cited on page 91]

-
- [254] A. Armando and S. E. Ponta, “Model checking of security-sensitive business processes,” in *International Workshop on Formal Aspects in Security and Trust*, pp. 66–80, Springer, 2009. [Cited on page 92]
- [255] D. R. dos Santos and S. Ranise, “A survey on workflow satisfiability, resiliency, and related problems,” *CoRR*, *abs/1706.07205*, 2017. [Cited on page 92]
- [256] W. M. van Der Aalst, A. H. Ter Hofstede, B. Kiepuszewski, and A. P. Barros, “Workflow patterns,” *Distributed and parallel databases*, vol. 14, no. 1, pp. 5–51, 2003. [Cited on page 93]
- [257] J. Crampton and G. Gutin, “Constraint expressions and workflow satisfiability,” in *Proceedings of the 18th ACM symposium on Access control models and technologies*, pp. 73–84, 2013. [Cited on page 93]
- [258] J. Holderer, R. Accorsi, and G. Müller, “When four-eyes become too much: a survey on the interplay of authorization constraints and workflow resilience,” in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pp. 1245–1248, 2015. [Cited on page 93]
- [259] Q. Wang and N. Li, “Satisfiability and resiliency in workflow authorization systems,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 4, pp. 1–35, 2010. [Cited on page 93]
- [260] J. C. Mace, C. Morisset, and A. Van Moorsel, “Quantitative workflow resiliency,” in *European Symposium on Research in Computer Security*, pp. 344–361, Springer, 2014. [Cited on page 93]
- [261] J. Crampton, G. Gutin, and D. Karapetyan, “Valued workflow satisfiability problem,” in *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, pp. 3–13, 2015. [Cited on page 93]
- [262] C. Bertolissi, D. R. Dos Santos, and S. Ranise, “Solving multi-objective workflow satisfiability problems with optimization modulo theories techniques,” in *Proceedings of the 23rd ACM on Symposium on Access Control Models and Technologies*, pp. 117–128, 2018. [Cited on page 93]
- [263] J. Crampton, G. Gutin, and R. Watrigant, “On the satisfiability of workflows with release points,” in *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*, pp. 207–217, 2017. [Cited on page 93]

- [264] T. Grimm, D. Lettnin, and M. Hübner, “A survey on formal verification techniques for safety-critical systems-on-chip,” *Electronics*, vol. 7, no. 6, p. 81, 2018. [Cited on pages 94 and 99]
- [265] W. McCune, “Otter 3.3 reference manual,” *arXiv preprint cs/0310056*, 2003. [Cited on page 94]
- [266] S. Owre, J. M. Rushby, and N. Shankar, “Pvs: A prototype verification system,” in *International Conference on Automated Deduction*, pp. 748–752, Springer, 1992. [Cited on page 94]
- [267] D. Mentré, C. Marché, J.-C. Filiâtre, and M. Asuka, “Discharging proof obligations from atelier b using multiple automated provers,” in *International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, pp. 238–251, Springer, 2012. [Cited on page 94]
- [268] J.-R. Abrial and J.-R. Abrial, *The B-book: assigning programs to meanings*. Cambridge University Press, 2005. [Cited on page 94]
- [269] F. Wiedijk, *The seventeen provers of the world: Foreword by Dana S. Scott*, vol. 3600. Springer, 2006. [Cited on page 94]
- [270] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, “Nusmv 2: An opensource tool for symbolic model checking,” in *International conference on computer aided verification*, pp. 359–364, Springer, 2002. [Cited on pages xii and 95]
- [271] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, “The nuxmv symbolic model checker,” in *International Conference on Computer Aided Verification*, pp. 334–342, Springer, 2014. [Cited on page 96]
- [272] G. Behrmann, A. David, and K. G. Larsen, “A tutorial on uppaal,” *Formal methods for the design of real-time systems*, pp. 200–236, 2004. [Cited on page 96]
- [273] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, “Symbolic model checking without bdds,” in *International conference on tools and algorithms for the construction and analysis of systems*, pp. 193–207, Springer, 1999. [Cited on page 96]

- [274] E. Clarke, D. Kroening, and F. Lerda, “A tool for checking ansi-c programs,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 168–176, Springer, 2004. [Cited on page 97]
- [275] R. Mukherjee, D. Kroening, and T. Melham, “Hardware verification using software analyzers,” in *2015 IEEE Computer Society Annual Symposium on VLSI*, pp. 7–12, IEEE, 2015. [Cited on page 97]
- [276] R. Sebastiani, “Lazy satisfiability modulo theories,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 3, no. 3-4, pp. 141–224, 2007. [Cited on page 97]
- [277] R. Nieuwenhuis and A. Oliveras, “On sat modulo theories and optimization problems,” in *International conference on theory and applications of satisfiability testing*, pp. 156–169, Springer, 2006. [Cited on page 98]
- [278] A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani, and C. Stenico, “Satisfiability modulo the theory of costs: Foundations and applications,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 99–113, Springer, 2010. [Cited on page 98]
- [279] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik, “Symbolic optimization with smt solvers,” *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 607–618, 2014. [Cited on page 98]
- [280] N. Bjørner, A.-D. Phan, and L. Fleckenstein, “vz-an optimizing smt solver,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 194–199, Springer, 2015. [Cited on page 98]
- [281] D. Chatterjee and V. Bertacco, “Equipe: Parallel equivalence checking with gp-gpus,” in *2010 IEEE International Conference on Computer Design*, pp. 486–493, IEEE, 2010. [Cited on page 98]
- [282] C. Van Eijk, “Sequential equivalence checking based on structural similarities,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 7, pp. 814–819, 2000. [Cited on page 98]
- [283] C. I. C. Marquez, M. Strum, and W. J. Chau, “Formal equivalence checking between high-level and rtl hardware designs,” in *2013 14th Latin American Test Workshop-LATW*, pp. 1–6, IEEE, 2013. [Cited on page 99]

- [284] P. Cousot and R. Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, 1977. [Cited on page 99]
- [285] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, “Klee: unassisted and automatic generation of high-coverage tests for complex systems programs,” in *OSDI*, vol. 8, pp. 209–224, 2008. [Cited on page 99]
- [286] P. Godefroid, M. Y. Levin, D. A. Molnar, *et al.*, “Automated whitebox fuzz testing,” in *NDSS*, vol. 8, pp. 151–166, 2008. [Cited on page 99]
- [287] D. Kästner, X. Leroy, S. Blazy, B. Schommer, M. Schmidt, and C. Ferdinand, “Closing the gap—the formally verified optimizing compiler compcert,” in *SSS’17: Safety-critical Systems Symposium 2017*, pp. 163–180, CreateSpace, 2017. [Cited on page 99]
- [288] H. Foster, *Applied assertion-based verification: An industry perspective*. Now Publishers Inc, 2009. [Cited on page 100]
- [289] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, “Formal methods: Practice and experience,” *ACM computing surveys (CSUR)*, vol. 41, no. 4, pp. 1–36, 2009. [Cited on page 100]
- [290] G. Dodig-Crnkovic, “Scientific methods in computer science,” in *Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden, Skövde, Suecia*, pp. 126–130, 2002. [Cited on page 103]
- [291] C. R. Kothari, *Research methodology: Methods and techniques*. New Age International, 2004. [Cited on page 103]
- [292] R. Kumar, *Research methodology: A step-by-step guide for beginners*. Sage, 2018. [Cited on page 104]
- [293] A. Cimatti, M. Dorigatti, and S. Tonetta, “Odra: A tool for checking the refinement of temporal contracts,” in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 702–705, IEEE, 2013. [Cited on pages 112 and 135]

- [294] M. Gario and A. Micheli, “PySMT: a solver-agnostic library for fast prototyping of smt-based algorithms,” in *SMT workshop*, vol. 2015, 2015. [Cited on pages 113 and 187]
- [295] D. Brand, “Verification of large synthesized designs,” in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pp. 534–537, IEEE, 1993. [Cited on page 124]
- [296] D. Harris and S. Harris, *Digital design and computer architecture*. Morgan Kaufmann, 2010. [Cited on page 153]
- [297] R. Rudell, “Dynamic variable ordering for ordered binary decision diagrams,” in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pp. 42–47, IEEE, 1993. [Cited on pages 177 and 203]
- [298] C. Barrett, A. Stump, C. Tinelli, *et al.*, “The smt-lib standard: Version 2.0,” in *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, vol. 13, p. 14, 2010. [Cited on page 187]
- [299] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Van Rossum, S. Schulz, and R. Sebastiani, “The mathsat 3 system,” in *International Conference on Automated Deduction*, pp. 315–321, Springer, 2005. [Cited on page 187]
- [300] F. Somenzi, “Cudd: Cu decision diagram package-release 2.4. 0,” *University of Colorado at Boulder*, 2012. [Cited on page 188]
- [301] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008. [Cited on page 188]
- [302] A. Hagberg and D. Conway, “Networkx: Network analysis with python.” [Cited on page 188]
- [303] K. S. Brace, R. L. Rudell, and R. E. Bryant, “Efficient implementation of a bdd package,” in *27th ACM/IEEE design automation conference*, pp. 40–45, IEEE, 1990. [Cited on page 189]
- [304] A. Rauzy, “Mathematical foundations of minimal cutsets,” *IEEE Transactions on Reliability*, vol. 50, no. 4, pp. 389–396, 2001. [Cited on page 190]

- [305] J. A. Abraham and D. P. Siewiorek, "An algorithm for the accurate reliability evaluation of triple modular redundancy networks," *IEEE Transactions on Computers*, vol. 100, no. 7, pp. 682–692, 1974. [Cited on page 191]
- [306] B. Bollig and I. Wegener, "Improving the variable ordering of obdds is np-complete," *IEEE Transactions on computers*, vol. 45, no. 9, pp. 993–1002, 1996. [Cited on page 203]
- [307] M. Fujita, Y. Matsunaga, and T. Kakuda, "On variable ordering of binary decision diagrams for the application of multi-level logic synthesis," in *Proceedings of the European Conference on Design Automation.*, pp. 50–54, IEEE, 1991. [Cited on page 203]
- [308] M. Beccuti, A. Bobbio, G. Franceschinis, and R. Terruggia, "A new symbolic approach for network reliability analysis," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pp. 1–12, IEEE, 2012. [Cited on pages xx and 218]
- [309] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pp. 28–28, IEEE, 1998. [Cited on page 225]
- [310] A. Kloeckner, "Pymetis: A python wrapper for metis." <https://mathematician.de/software/pymetis/>, 2007–2020. [Cited on page 235]
- [311] D. W. Coit and A. E. Smith, "Reliability optimization of series-parallel systems using a genetic algorithm," *IEEE Transactions on reliability*, vol. 45, no. 2, pp. 254–260, 1996. [Cited on page 250]
- [312] Y.-C. Liang and A. E. Smith, "An ant system approach to redundancy allocation," in *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, vol. 2, pp. 1478–1484, IEEE, 1999. [Cited on page 250]
- [313] D. W. Coit and J. C. Liu, "System reliability optimization with k-out-of-n subsystems," *International Journal of Reliability, Quality and Safety Engineering*, vol. 7, no. 02, pp. 129–142, 2000. [Cited on page 250]
- [314] D. W. Coit and A. Konak, "Multiple weighted objectives heuristic for the redundancy allocation problem," *IEEE transactions on reliability*, vol. 55, no. 3, pp. 551–558, 2006. [Cited on page 250]

- [315] R. Tavakkoli-Moghaddam, J. Safari, and F. Sassani, "Reliability optimization of series-parallel systems with a choice of redundancy strategies using a genetic algorithm," *Reliability Engineering & System Safety*, vol. 93, no. 4, pp. 550–556, 2008. [Cited on page 250]
- [316] A. Chambari, S. H. A. Rahmati, A. A. Najafi, *et al.*, "A bi-objective model to optimize reliability and cost of system with a choice of redundancy strategies," *Computers & Industrial Engineering*, vol. 63, no. 1, pp. 109–119, 2012. [Cited on page 250]
- [317] J. Safari, "Multi-objective reliability optimization of series-parallel systems with a choice of redundancy strategies," *Reliability Engineering & System Safety*, vol. 108, pp. 10–20, 2012. [Cited on page 250]
- [318] M. A. Ardakan and A. Z. Hamadani, "Reliability optimization of series-parallel systems with mixed redundancy strategy in subsystems," *Reliability Engineering & System Safety*, vol. 130, pp. 132–139, 2014. [Cited on page 250]
- [319] M. A. Ardakan, A. Z. Hamadani, and M. Alinaghian, "Optimizing bi-objective redundancy allocation problem with a mixed redundancy strategy," *ISA transactions*, vol. 55, pp. 116–128, 2015. [Cited on page 250]
- [320] H. Gholinezhad and A. Z. Hamadani, "A new model for the redundancy allocation problem with component mixing and mixed redundancy strategy," *Reliability Engineering & System Safety*, vol. 164, pp. 66–73, 2017. [Cited on page 250]
- [321] A. E. Jahromi and M. Feizabadi, "Optimization of multi-objective redundancy allocation problem with non-homogeneous components," *Computers & Industrial Engineering*, vol. 108, pp. 111–123, 2017. [Cited on page 250]
- [322] H. Kim, "Maximization of system reliability with the consideration of component sequencing," *Reliability Engineering & System Safety*, vol. 170, pp. 64–72, 2018. [Cited on page 250]
- [323] A. Peiravi, M. Karbasian, M. A. Ardakan, and D. W. Coit, "Reliability optimization of series-parallel systems with k-mixed redundancy strategy," *Reliability Engineering & System Safety*, vol. 183, pp. 17–28, 2019. [Cited on page 250]

- [324] P. P. Guilani, M. N. Juybari, M. A. Ardakan, and H. Kim, “Sequence optimization in reliability problems with a mixed strategy and heterogeneous backup scheme,” *Reliability Engineering & System Safety*, vol. 193, p. 106660, 2020. [Cited on page 250]
- [325] M. Reihaneh, M. A. Ardakan, and M. Eskandarpour, “An exact algorithm for the redundancy allocation problem with heterogeneous components under the mixed redundancy strategy,” *European Journal of Operational Research*, vol. 297, no. 3, pp. 1112–1125, 2022. [Cited on page 250]
- [326] P. Nuzzo, H. Xu, N. Ozay, J. B. Finn, A. L. Sangiovanni-Vincentelli, R. M. Murray, A. Donzé, and S. A. Seshia, “A contract-based methodology for aircraft electric power system design,” *IEEE Access*, vol. 2, pp. 1–25, 2013. [Cited on page 264]
- [327] N. Bajaj, P. Nuzzo, M. Masin, and A. Sangiovanni-Vincentelli, “Optimized selection of reliable and cost-effective cyber-physical system architectures,” in *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 561–566, IEEE, 2015. [Cited on page 264]
- [328] “Python software foundation. Python language referen, version 3.9. Available at <https://www.python.org/>.” [Cited on page 309]
- [329] “PySMT, a solver-agnostic library for SMT Formulae manipulation and solving. Available at <https://pysmt.org/>.” [Cited on page 309]
- [330] “Networkx, a software for complex networks. Available at <https://networkx.org/>.” [Cited on pages 309 and 314]
- [331] “Pycharm, the Python IDE for Professional Developers. Available at: <https://www.jetbrains.com/pycharm/>.” [Cited on page 311]
- [332] “Metis, a family of graph and hypergraph partitioning software. Available at <http://glaros.dtc.umn.edu/gkhome/views/metis/>.” [Cited on page 314]

Appendix A

Software Dependency Graph

Our software tool is available in the following repository:

<https://github.com/mistert0974/MORA.git>

In the following, we report the dependency graph, and provide a brief description of the dependencies of Python modules.

A.1 Software structure

A dependency graph is a data structure formed by a directed graph that describes the dependency of an entity in the system on the other entities of the same system. Each node of the graph represent a software module, and it points to the node (module) on which it depends.

The blue block in Figure A.1 represents the set of components composing the redundant systems we deal with. You can find the sub-components of the redundant patterns (like modules and voters), and stage, concretizer and abstractor needed to build the CSAs of the Miter composition. The orange block in A.1 represents the library including the templates of all redundant patterns.

For the experimental evaluation, we built a library of examples and benchmarks. In each test example file, the basic system architecture is defined, and

the library of redundant patterns available for each component is specified. Then, the optimizer is invoked, choosing one of the three approaches available for optimization: enumerative, symbolic, or hybrid.

Design objectives are specified in the module named "params.py". The module "rel_tools.py" is the module where all formulae are built, while "rel_extractor.py" implements the algorithm described in Section 6.3.7.

The module named "arch_node.py" contains the Class we created to manage the nodes of the architectures. It includes the methods to define configuration atoms, fault atoms, the CSA for each pair (*Component, Pattern*), the linking constraints, compatibility constraints, and to perform the quantifier elimination to the entire architecture, retrieve the Boolean formula describing the configuration, and map the probability to the fault atoms.

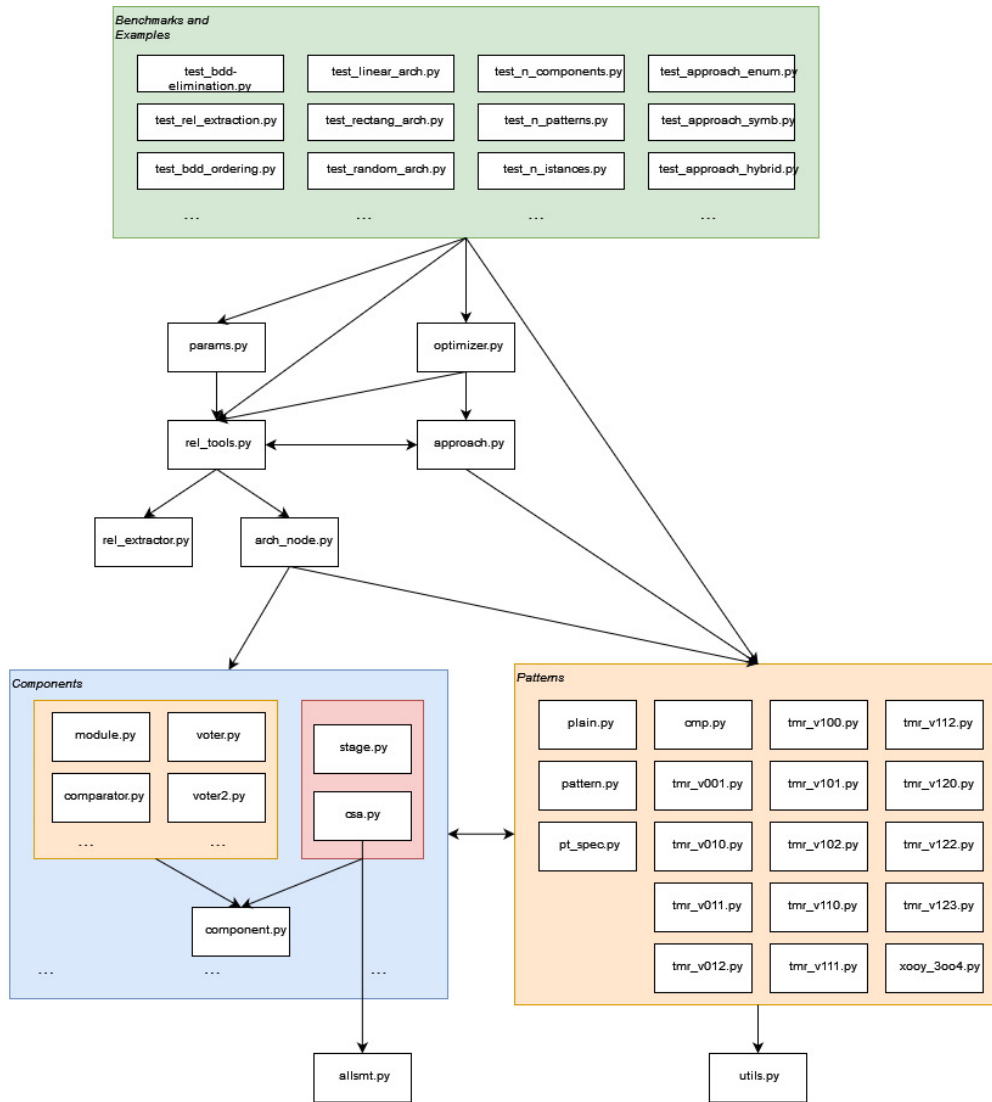


Figure A.1: Dependency graph of the software tool implementing the proposed method

Appendix B

Installation of Required Tools

In the following, we report a step-by-step guide for the installation of required software tools and packages. It can help you avoid common pitfalls and errors that can occur during software installation.

B.1 Software needed

The simulations were run under Ubuntu 20.04 operating system. To use the tool implementing the method proposed in this work, the following software tools and packages are needed:

- Python [328]
- PySMT [329]
- Solvers
- Networkx [330]
- Plotly
- Matplotlib
- Pygraphviz

- Pandas
- Metis

B.2 Installation Procedure

This section provides a step-by-step installation procedure of our software tool.

B.2.1 Installing Python

Python is an interpreted, object-oriented, high-level programming language. We chose it because of its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for rapid application development, as well as for use as a scripting or glue language to connect existing components together. It has an easy to learn syntax, and supports modules and packages, which encourages program modularity and code reuse. And It can be freely distributed. Most versions of Ubuntu (including the 20.04 we used) come with Python pre-installed. Check your version of Python by entering the following:

```
1 $ python3 --version
```

You can then update it by using the following commands (as illustrated in Figure B.1).

```
1 $ sudo apt update
```

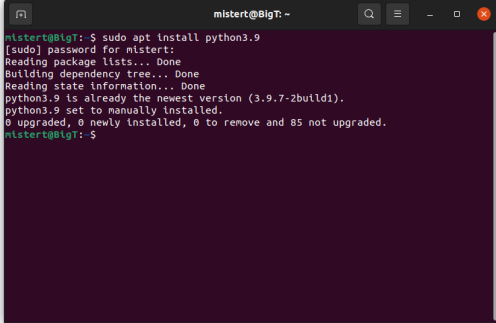
A terminal window titled 'mistert@BigT: ~' showing the command 'sudo apt install python3.9' being executed. The output shows the system checking for updates, reading package lists, building a dependency tree, and reading state information. It then reports that python3.9 is already the newest version (3.9.7-2build1) and is set to manually installed. The terminal ends with the prompt 'mistert@BigT:~\$'.

Figure B.1: Installing Python

Otherwise, you can install it by using the following command:

```
1 $ sudo apt install python3.9
```

B.2.2 Installing Python IDE [OPT]

We suggest that you install an, which provides comprehensive facilities for software development. We employed Pycharm [331], but you can use the one you prefer. You can install it from command line:

```
1 $ sudo snap install pycharm-community --classic
```

Or using the graphical user interface provided by Ubuntu Software Center (as illustrated in Figure B.2).

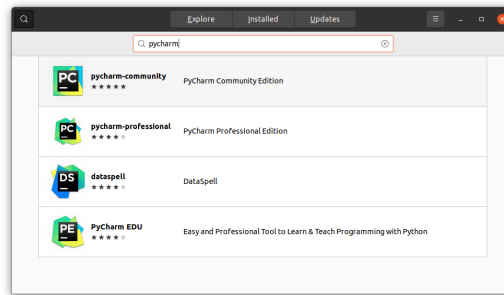


Figure B.2: Installing Pycharm

B.2.3 Installing pySMT

The package pySMT is a Python API that eases the work with SMT. You can install it via the package management tool named *pip* as follows:

```
1 $ sudo apt install python3-pip
2 $ pip install PySMT
```

As indicated in Figure B.3 remember to add the installation folder to the *path* variable (see Figure B.4)

B.2.4 Installing Solvers

PySMT works with any SMT-LIB compatible solver, such as MathSAT, Z3, and CUDD, which can be installed by using the following commands:

```
mistert@Big: ~$ pip install pysmt
Collecting pysmt
  Using cached PysMT-0.9.0-py2.py3-none-any.whl (317 kB)
Requirement already satisfied: six in /usr/lib/python3/dist-packages (from pysmt) (1.16.0)
Installing collected packages: pysmt
  WARNING: The script pysmt-install is installed in '/home/mistert/.local/bin' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed pysmt-0.9.0
mistert@Big: ~$
```

Figure B.3: Installing PySMT

```
111 if ! shopt -oq posix; then
112   if [ -f /usr/share/bash-completion/bash_completion ]; then
113     . /usr/share/bash-completion/bash_completion
114   elif [ -f /etc/bash_completion ]; then
115     . /etc/bash_completion
116   fi
117 fi
118 export PATH="/home/mistert/.local/bin:$PATH"
119
```

Figure B.4: Adding PySMT to Path

```
1 $ pysmt install --msat
2 $ pysmt install --z3
3 $ pysmt install --bdd
```

By default, the solvers are downloaded, unpacked and built in your home directory in the `.smt_solvers` folder. Compiled libraries and actual solver packages are installed in the relevant site-packages directory (e.g. virtual environment's packages root or local user-site).

When installing MatSAT solver, you may encounter the following error:

```
../include/mathsat.h:32:10: fatal error: gmp.h: File o directory non esistente
```

You need to install first the multiprecision arithmetic library named `libgmp3-dev`, by using the following commands:

```
1 $ sudo apt-get update -y
2 $ sudo apt-get install -y libgmp3-dev
```

Afterwards, the installation of MatSAT via pySMT is successful.

Furthermore, at the moment of writing, to use MathSAT in Optimization mode (i.e., OptiMathSAT) via pySMT, we had to install it manually, as we had to use the Github branch named "optimization" (not merged in the master yet). Extract the branch to a temporary folder, then from that folder, run the `setup.py` using the direct path to the `virtualenv` python instance:

```
1 $ /home/antonio/PycharmProjects/DSE_001/venv/bin/python3 setup.py install
```

If you had not created a *virtualenv* yet, please note that the main purpose of virtual environments is to manage settings and dependencies of a particular project regardless of other Python projects. This means that you can create a project-specific isolated virtual environment. If you use PyCharm (or some other IDE supporting it), you do not need to install a virtual environment manually, because PyCharm makes it possible to use its *virtualenv* tool, as illustrated in Figure B.5).

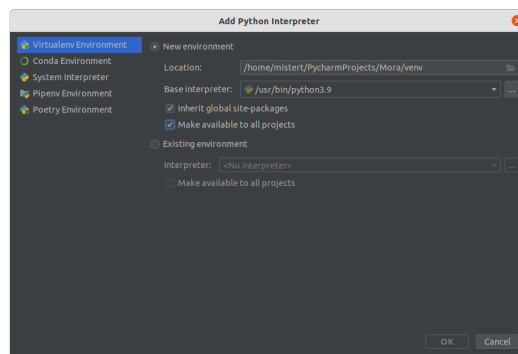


Figure B.5: Installing a virtual environment via PyCharm IDE.

Also CUDD Solver created some problems. To be precise, pySMT uses RepyCUDD, a python wrapper for the CUDD BDD library. This is a fork of PyCUDD (<http://bears.ece.ucsb.edu/pycudd.html>). The main purpose of this fork is to provide pySMT a re-entrant wrapper for CUDD, so that some features (like ADD and ZDD) not currently used by PySMT could be broken in RepyCUDD. We encountered some issues when installing CUDD Solver via pySMT due to the missing installation of Swig, a software development tool that gives script language like Python the ability to invoke C/C++ libraries. Be sure to install Swig before installing the CUDD solver via pySMT. Instructions to install Swig on Ubuntu follow.

```
1 $ sudo apt-get update -y
2 $ sudo apt-get install -y swig
```

You can check the solvers installed by using the following command (see Figure B.6):

```

mistert@BigT: ~
└─$ make
make[1]: Leaving directory '/home/mistert/.smt_solvers/bdd/repycudd-ecb03d6d231273343178f566cc4d7258dce52b4/cudd-2.4.2'
g++ -shared repycudd.o repycudd_wrap.o -xlinker -rpath /home/mistert/.smt_solvers/bdd/repycudd-ecb03d6d231273343178f566cc4d7258dce52b4/cudd-2.4.2//lib -o repycudd.so -L/home/mistert/.smt_solvers/bdd/repycudd-ecb03d6d231273343178f566cc4d7258dce52b4/cudd-2.4.2//lib -ln -lstdc++ -lcudd -lcuddntr -lcuddst -lcudduttl -lddnp -lcuddepd;
make: Leaving directory '/home/mistert/.smt_solvers/bdd/repycudd-ecb03d6d231273343178f566cc4d7258dce52b4'
mistert@BigT: ~
└─$ pysmt-install --check
Installed Solvers:
msat      True (5.6.1)
cvc4      False (None)
z3        True (4.8.7)
yices     False (None)
btor      False (None)
picosat   False (None)
bdd       True (2.0.3)

Solvers: z3, msat, bdd
Quantifier Eliminator: z3, msat_fm, msat_lw, bdd, shannon, selfsub
UNSAT-cores: z3, msat
Interpolators: msat
mistert@BigT: ~
└─$

```

Figure B.6: Installed solver for pySMT

B.2.5 Installing other useful packages

Our software tool leverage some other useful packages, that can be easily installed with *pip* or directly from Pycharm IDE.

- **Networkx** [330]: It is a Python package for the management of complex networks. We used it to model and manipulate the system architecture.
- **Plotly**: a graphing library for Python
- **Matplotlib**: a library for creating visualizations in Python
- **Pygraphviz**: a Python package to create and edit graphs.
- **Pandas**: a open source tool for data analysis and manipulation, built on top Python.
- **pyMetis**: a Python wrapper for the Metis graph partitioning software.

Pleas note that METIS [332] itself is not included with the pyMetis wrapper. You need to install it. The wrapper was designed for use with METIS 5 (we used the version 5.1.0), downloadable at the folloeing url:

<http://glaros.dtc.umn.edu/gkhome/metis/metis/download>.

These are some preliminary operations before installing the 5.0 release of METIS. You need to need to have GNU make and CMake installed. Install GNU make by running:

¹ `$ sudo apt-get install build-essential`

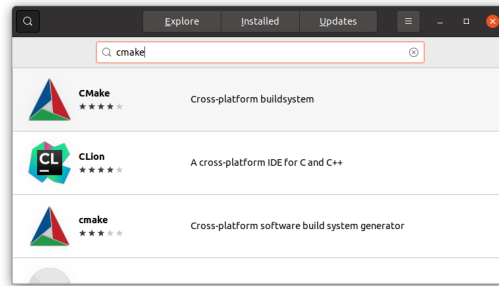


Figure B.7: Installation of CMake

You can install CMake via the Ubuntu software manager:

Then, edit the file `include/metis.h` and specify the width (32 or 64 bits) of the elementary data type used in METIS. This is controlled by the `IDXTYPEWIDTH` constant. On a 32 bit architecture you can only specify a width of 32, whereas for a 64 bit architecture you can specify a width of either 32 or 64 bits.

From the Metis directory execute the following commands.

```
1 $ make config
2 $ make
```

Furthermore, the shared library is needed, and it is not enabled by default by the configuration process. Turn it on with the following command:

```
1 $ make config shared=1
```

Hence:

```
1 $ sudo make install
```

If you cannot get `sudo` rights with your user, you cannot write to `/usr/local/bin`. However, that might not even be necessary, as you can also install programs somewhere else, and amend your `$PATH` environment variable, which tells your shell where it can find executable programs.

In addition, the wrapper uses a few environment variables.

- `METIS_DLL`: probably, Python will raise the following run-time error: `'Could not load METIS dll: libmetis.so'`. That is because Python is unable to automatically locate the shared library, You can easily fix it by setting the full path to it in this environment variable:

```
1      $ export METIS_DLL=/usr/local/lib/libmetis.so
```

One last note: you can still get an error as follows: 'OSError: libmetis.so: cannot open shared object file: No such file or directory'. You can fix it by running:

```
1      $ ldconfig
```

Since a shared library can be used by others, this command is used to create required links and cache and manage them.

- *METIS_IDXTYPEWIDTH*: see below.
- *METIS_REALTYPEWIDTH*: the sizes of the *idx_t* and *real_t* types are not easily determinable at run-time, so they can be provided with these environment variables.