Pós-Graduação em Ciência da Computação

André Luís Ribeiro Didier

An Algebra of Temporal Faults

Ph.D. Thesis

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
<http://www.cin.ufpe.br/~posgraduacao>

Recife, PE
2016

André Luís Ribeiro Didier

An Algebra of Temporal Faults

A Ph.D. Thesis presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Philosophy Doctor in Computer Science.

Federal University of Pernambuco

Center of Informatics

Graduate in Computer Science

Supervisor: Alexandre Cabral Mota

Co-supervisor: Alexander Romanovsky

Recife, PE

2016

André Luís Ribeiro Didier
An Algebra of Temporal Faults/ André Luís Ribeiro Didier– Recife, PE, 2016-
162 p. : il.(alguma color.); 30 cm.

Supervisor: Alexandre Cabral Mota

Co-supervisor: Alexander Romanovsky

Ph.D. Thesis – Federal University of Pernambuco
Center of Informatics
Graduate in Computer Science, 2016.

1. Fault Trees. 2. Dependability. 3. Fault Tolerance. 4. Fault Removal. I. Alexandre Cabral Mota II. Alexander Romanovsky III. Universidade Federal de Pernambuco. IV. Centro de Informática. V. Título

CDU 02:141:005.7

André Luís Ribeiro Didier

An Algebra of Temporal Faults

A Ph.D. Thesis presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Philosophy Doctor in Computer Science.

Prof. Augusto Cesar Alves Sampaio
Centro de Informática/UFPE

Prof. Paulo Romero Martins Maciel
Centro de Informática/UFPE

Prof. Enrique Andrés López Droguett
Departamento de Engenharia de
Produção/UFPE

Recife, PE
2016

I dedicate this thesis to Juliana, Luciana (pipoquinha), and Bianca (snowflake).

Acknowledgments

If I were afraid of the path, I wouldn't have gotten here.

Two men helped me to build this path far before I started my scholar journey: Roberto and Júnior. My two grandfathers couldn't see how far I got. My heart was with them all the time, but I was physically far away from them in their very last breath. May God have them in his arms.

It is now eleven years since I graduated. I met professors Alexandre and Augusto still during the Computing Science undergrad course. They have been present in my academic life ever since. Their comments, instructions, talks, (even jokes), are what molded my path to here. I have no words to express how much I thank them, specially Alexandre.

CNPq and FACEPE were keen to guarantee my existential needs. The former with the trip to Newcastle upon Tyne, and the latter during the time I stayed in Recife, before and after the trip.

I thank to Sascha Romanovsky for accepting me as his advisee while I was a Research Assistant of the COMPASS project. His comments, instructions, and knowledge were of great importance for this work.

My stay in Newcastle upon Tyne couldn't be as good as it was without the hospitality, useful discussions, and support of my colleagues at Newcastle University. A big THANK YOU to John Fitzgerald, Zoe Andrews, Richard Payne, Claire Smith, Dee Carr, Claire Ingram, my shared office colleague Anirban Bhattacharyya, and all other staff members.

Still in Newcastle upon Tyne, I thank all friends my family and I made outside University. Thanks to Kelechi Dibie and her family to welcome us for the Christmas' and new year's dinners. They were our family abroad.

I thank all my family for their patience to have me away in several family reunions, due the time required to do this work. In special, my two girls and my wife.

“Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity.

(Alan Turing)

Resumo

A modelagem de falhas é essencial na antecipação de defeitos em sistemas críticos. Tradicionalmente, Árvores de Falhas Estáticas são empregadas para este fim, mas Árvores de Falhas Temporais e Dinâmicas têm ganhado evidência devido ao seu maior poder para modelar e detectar propagações complexas de falhas que levam a um defeito.

Em um trabalho anterior, mostramos uma estratégia baseada na álgebra de processos CSP e modelos Simulink para obter rastros (sequências) de falhas que levam a um defeito. A partir dos rastros de falhas nós descartamos a informação de ordenamento para obter expressões de estrutura para Ávores de Falhas Estáticas. Ao contrário de descartar tal informação de ordenamento, poderíamos usá-la para obter expressões de estrutura para Árvores de Falhas Temporais ou Dinâmicas.

No presente trabalho apresentamos: (i) uma álgebra temporal de falhas (com noção de propagação de falhas) para analisar defeitos em sistemas e provamos que ela é de fato uma álgebra Booleana, e (ii) uma lógia de ativação parametrizada para expressar comportamentos nominais e de falha, incluindo a modelagem de falhas a partir de uma álgebra e um conjunto de modos de operação. A álgebra permite herdar as propriedades de álgebras Booleanas, leis e técnicas de redução existentes, as quais são muito benéficas para a modelagem e análise de falhas. Com expressões na álgebra temporal de falhas nós permitimos a verificação de propriedades de segurança (*safety*) baseadas em Árvores de Falhas Estáticas, Temporais ou Dinâmicas. A lógica criada neste trabalho pode ser usada com outras álgebras além das apresentadas. Sendo usada em conjunto com a álgebra temporal de falhas, tem a intenção de ajudar os analistas a considerar todas as possíveis situações em expressões complexas com operadores relacionados ao ordenamento das falhas, evitando esquecer combinações de falhas sutis (porém relevantes). Além disso, nossa álgebra temporal de falhas trata operadores NOT, que têm sido deixados de fora em outros trabalhos. Nós ilustramos nosso trabalho com alguns estudos de caso simples, mas reais, fornecidos pelo nosso parceiro industrial, a EMBRAER.

Isabelle/HOL foi utilizado para a mecanização das provas dos teoremas da álgebra temporal de falhas.

Palavras-chave: Simulink, CSP, FDR, Fault Tree Analysis, Temporal Fault Trees, Dynamic Fault Trees, Isabelle/HOL, Pandora, Fault Injection

Abstract

Fault modelling is essential to anticipate failures in critical systems. Traditionally, Static Fault Trees are employed to this end, but Temporal and Dynamic Fault Trees have gained evidence due to their enriched power to model and detect intricate propagation of faults that lead to a failure.

In a previous work, we showed a strategy based on the process algebra CSP and Simulink models to obtain fault traces that lead to a failure. From the fault traces we discarded the ordering information to obtain structure expressions for Static Fault Trees. Instead of discarding such an ordering information, it could be used to obtain structure expressions of Temporal or Dynamic Fault Trees.

In this work we present: (i) an algebra of temporal faults (with a notion of fault propagation) to analyse systems' failures, and prove that it is indeed a Boolean algebra, and (ii) a parametrized activation logic to express nominal and erroneous behaviours, including fault modelling, provided an algebra and a set of operational modes. The algebra allows us to inherit Boolean algebra's properties, laws and existing reduction techniques, which are very beneficial for fault modelling and analysis. With expressions in the algebra of temporal faults we allow the verification of safety properties based on Static, Temporal or Dynamic Fault Trees. The logic created in this work can be combined with other algebra beyond those shown here. Being used with the algebra of temporal faults it is intended to help analysts to consider all possible situations in complex expressions with order-related operators, avoiding missing subtle (but relevant) faults combinations. Furthermore, our algebra of temporal faults tackles NOT operators which have been left out in other works. We illustrate our work on simple but real case studies, some supplied by our industrial partner EMBRAER.

Isabelle/HOL was used to mechanize the theorems proofs of the algebra of temporal faults.

Keywords: Simulink, CSP, FDR, Fault Tree Analysis, Temporal Fault Trees, Dynamic Fault Trees, Isabelle/HOL, Pandora, Fault Injection

List of figures

Figure 1 – Traditional Fault Tree Analysis (FTA) 28
Figure 2 – Faults injection and Algebra of Temporal Faults (ATF) to perform FTA 34
Figure 3 – Activation Logic (AL) and ATF to perform FTA 35
Figure 4 – Strategy overview 36
Figure 5 – Relation of two events with duration 42
Figure 6 – Static Fault Tree (SFT) symbols using a free commercial tool 46
Figure 7 – SFT symbols as in the Fault Tree Handbook 47
Figure 8 – SFT gates 48
Figure 9 – Very simple example of a fault tree 48
Figure 10 – TFT-specific gates 50
Figure 11 – TFT small example 50
Figure 12 – DFTs's original gates symbols 52
Figure 13 – Dynamic Fault Trees's (DFTs's) [1, 2] gates symbols 52
Figure 14 – DFT example 55
Figure 15 – A diagram for a truth table 57
Figure 16 – A BDD for the expression $A \vee (\neg B \wedge C)$ 57
Figure 17 – DT for variables X and Y 58
Figure 18 – DT for the formula $(X \wedge Y) \vee ((X < Y) \wedge Z)$ 59
Figure 19 – ZBDD example of combination set $\{a, b\}$ 60
Figure 20 – Non-coherent FT college student's example 63
Figure 21 – Gas detection system 64
Figure 22 – FT for a generic failure in the gas detection system 65
Figure 23 – *Coherent* FT for the most critical outcome of the gas detection system 66
Figure 24 – *Non-coherent* FT for the most critical outcome of the gas detection system 66
Figure 25 – Block diagram of the ACS provided by our industrial partner (nominal model) 67
Figure 26 – Internal diagram of the monitor component (Figure 25 (A)). 67
Figure 27 – Isabelle/HOL window, showing the basic symmetry theorem 73
Figure 28 – AL overview 89

List of tables

Table 1 – TTT of TFT's operators and sequence value numbers 49
Table 2 – TTT of a simple example 51
Table 3 – Dynamic Fault Tree (DFT) [1, 2] conversion to calculate probability of top-level event 53
Table 4 – Algebraic model of DFT gates with inputs A and B 54
Table 5 – Date-of-occurrence function for operators defined in [3] 54
Table 6 – Truth table for a formula outputs with three variables (A, B, and C) . . 57
Table 7 – Annotations table of the ACS provided by our industrial partner 71

List of abbreviations and acronyms

AADL	Architecture Analysis and Design Language p. 29
AL	Activation Logic pp. 15, 32–35, 89–95, 97, 100–102
AFP	archive of formal proofs p. 73
ATF	Algebra of Temporal Faults pp. 15, 24, 25, 32–35, 43, 59, 70, 75–78, 81, 84–86, 88, 90, 97, 98, 103, 107, 119, 134–136, 140, 141, 143, 145, 147, 149, 151, 153, 155, 157–161
BDD	Binary Decision Diagram pp. 15, 27, 30, 31, 43, 47, 53, 56–60, 62, 107
BN	Bayesian network p. 53
CML	COMPASS Modelling Language p. 40
CPN	coloured Petri-net p. 53
CSP	Communicating Sequential Processes p. 40
CSP$_M$	Communicating Sequential Processes pp. 30, 33, 43, 67, 68, 70, 75, 87
CTMC	continuous-time Markov chain pp. 30, 53
DBN	dynamic bayesian network p. 31
DD	Dependence Diagram p. 41
DFT	Dynamic Fault Tree pp. 17, 28–32, 37, 41, 43–45, 48, 49, 51–56, 59, 60, 70, 75, 81, 89, 107
DNF	disjunctive normal form pp. 44, 50, 53, 58, 60, 81
DRBD	Dynamic Reliability Block Diagram p. 41
DT	dependency tree pp. 30, 50, 51, 56, 58
DTMC	discrete-time Markov chain pp. 30, 41, 51, 56, 107
FBA	Free Boolean Algebra pp. 27, 30, 31, 43, 55, 60, 61, 73, 75–78, 98, 107
FDR	Failures and Divergences Refinement pp. 30, 67, 68, 70
FMEA	Failure Modes and Effects Analysis pp. 31, 41
FSM	Finite State Machine p. 56
FT	fault tree pp. 15, 27–33, 35, 37, 38, 40, 43–46, 48, 49, 54, 55, 62–66, 70, 75, 84, 87, 107
FTA	Fault Tree Analysis pp. 15, 27–35, 43–45, 64, 84
HCAS	cardiac assist system p. 53
HiP-HOPS	Hierarchically Performed Hazard Origin and Propagation Studies pp. 29–31, 38, 45, 70
HLPN	high-level Petri-net p. 56

HOL	higher-order logic p. 72
Isar	Intelligible semi-automated reasoning pp. 43, 72
ITL	Interval Temporal Logic p. 55
LTL	linear temporal logic p. 47
MCS	minimal cut set pp. 27, 44, 47, 50, 52
MCSeq	minimal cut sequence pp. 29, 33, 49, 50, 52, 53, 56, 58, 59, 84, 87
PN	Petri-net p. 39
ROBDD	Reduced Ordered Binary Decision Diagram pp. 56–58
SBDD	Sequential Binary Decision Diagram pp. 31, 53, 56, 60
SFT	Static Fault Tree pp. 15, 28–30, 32, 37, 41, 43–48, 50, 52, 54–56, 58–60, 62, 70, 75, 81, 89, 107
SoS	System of Systems pp. 32, 38
SWN	stochastic well-formed net p. 53
SysML	Systems Modelling Language pp. 32, 40
TFT	Temporal Fault Tree pp. 28–32, 37, 43–45, 47–56, 70, 75, 81, 89
TTT	Temporal Truth Table pp. 17, 30, 49, 58
UML	Unified Modelling Language p. 40
Z	Z Notation pp. 56, 72
ZBDD	Zero-suppresed Binary Decision Diagram pp. 56, 59

Fault tree gates

AND	$\wedge$. Used in SFT, TFT, and DFT. pp. 27, 28, 43, 46, 48–50, 53, 54, 60, 62, 63, 70, 71, 81–83, 86, 88
CSp	cold spare. Used in DFT. pp. 29, 44, 52, 54, 56, 60
FDEP	functional dependency. Used in DFT. pp. 29, 44, 51, 52, 54
IBefore	inclusive-before. Used in structure expressions of DFT. pp. 53, 54, 60
NIBefore	non-inclusive-before. Used in structure expressions of DFT. pp. 53, 54
NOT	$\neg$. Used in non-coherent trees. pp. 29, 30, 32, 43, 48, 62, 64, 78, 81, 84, 85
OR	$\vee$. Used in SFT, TFT, and DFT. pp. 27, 28, 43, 46, 48, 49, 53, 54, 62, 70, 82, 83, 87
PAND	priority-AND. Used in SFT, TFT, and DFT. pp. 28, 43, 44, 48–50, 52, 54, 55, 60, 84, 85
POR	priority-OR. Used in TFT. pp. 48–50, 53
SAND	simultaneous-AND. Used in TFT. pp. 48–51, 53
SEQ	sequence enforcing. Used in DFT. pp. 29, 44, 52, 54
SIMLT	simultaneous. Used in structure expressions of DFT. pp. 53, 54
WSp	warm spare. Used in DFT. pp. 29, 60
XBefore	exclusive-before. Proposed in this work. pp. 24, 25, 75–79, 81–84, 88, 97, 98, 107, 134, 135

Contents

1 INTRODUCTION . . . 27
1.1 Research questions . . . 30
1.2 Proposed solution . . . 32
1.3 Contributions . . . 34
1.4 Thesis organization . . . 35

2 BASIC CONCEPTS . . . 37
2.1 Systems, dependability, and fault modelling . . . 37
2.1.1 Systems . . . 37
2.1.2 Dependability . . . 38
2.1.3 Fault Modelling . . . 40
2.2 Time relation of fault events . . . 41

3 ANALYSIS AND TOOLS . . . 43
3.1 Fault Tree Analysis and structure expressions . . . 43
3.1.1 Static Fault Trees . . . 45
3.1.2 Temporal Fault Trees . . . 47
3.1.3 Dynamic Fault Trees . . . 51
3.2 Structure expression analysis . . . 54
3.2.1 Stateful methods and temporal logic analysis . . . 55
3.2.2 Binary Decision Diagrams . . . 56
3.2.3 Dependency tree . . . 58
3.2.4 Zero-suppresed Binary Decision Diagrams . . . 59
3.2.5 Sequential Binary Decision Diagrams . . . 60
3.3 Free Boolean Algebras . . . 60
3.4 Using the NOT operator in static fault trees . . . 62
3.4.1 Non-coherent fault tree misleads . . . 63
3.4.2 Usefulness of NOT gates in FTA . . . 64
3.5 Systems nominal model and fault injection . . . 66
3.6 Isabelle/HOL . . . 72

4 A FREE ALGEBRA TO EXPRESS STRUCTURE EXPRESSIONS OF ORDERED EVENTS . . . 75
4.1 Temporal properties (*tempo*) . . . 79
4.2 XBefore laws . . . 81
4.3 Qualitative and quantitative properties . . . 84

4.3.1 Formal acceptance criteria . . . 87
4.4 Mapping Communicating Sequential Processes[1] (CSP$_M$) traces to Algebra of Temporal Faults (ATF) . . . 87

5 REASONING FAULT ACTIVATION . . . 89
5.1 The Activation Logic Grammar . . . 90
5.2 Healthiness Conditions . . . 91
5.2.1 H #1: No predicate is a contradiction . . . 92
5.2.2 H #2: All possibilities are covered . . . 92
5.2.3 H #3: There are no two terms with exactly the same operational mode. . . 93
5.2.4 Healthy expression . . . 93
5.3 Non-determinism . . . 94
5.4 Predicates Notation . . . 94

6 CASE STUDY . . . 97
6.1 From traces to structure expressions with Boolean operators . . . 97
6.2 From traces to structure expressions with XBefore . . . 98
6.3 From Activation Logic (AL) to structure expressions with Boolean operators . . . 100
6.4 From Activation Logic (AL) to structure expressions with exclusive-before (XBefore) . . . 103

7 CONCLUSION . . . 107
7.1 Future work . . . 107

BIBLIOGRAPHY . . . 109

APPENDIX 117

APPENDIX A – FORMAL PROOFS IN ISABELLE/HOL . . . 119
A.1 Sliceable . . . 119
A.1.1 Disjoint elements and sliceable . . . 120
A.1.2 n-th element in a sliceable . . . 120
A.1.3 Theorems for sliceable . . . 120
A.2 Sliceable distinct lists . . . 124
A.2.1 Properties of sliceable distinct lists . . . 127
A.3 Algebra of Temporal Faults . . . 134
A.3.1 Basic Algebra of Temporal Faults (ATF) operators and $\mathbf{tempo}_1$ . . . 134
A.3.2 Definition of associativity of exclusive-before (XBefore) . . . 134

---

[1] This variant "M" is the machine-readable version of CSP.

A.3.3 Equivalences in the ATF and properties . . . 134
A.3.4 XBefore transitivity . . . 135
A.3.5 Mixed operators in ATF . . . 135
A.3.6 Theorems in the context of ATF . . . 136
A.4 Denotational semantics for ATF . . . 140
A.4.1 Formula: distinct lists . . . 140
A.4.1.1 Formula as Boolean algebra . . . 140
A.4.1.2 Tempo properties . . . 141
A.4.1.3 Tempo properties for list member . . . 143
A.4.1.4 Tempo properties for other operators . . . 144
A.4.2 XBefore of distinct lists . . . 144
A.4.2.1 XBefore and temporal properties . . . 145
A.4.2.2 XBefore and appending . . . 145
A.4.2.3 XBefore, bot, top and idempotency . . . 145
A.4.2.4 XBefore neutral . . . 146
A.4.2.5 XBefore associativity . . . 147
A.4.2.6 XBefore equivalences . . . 147
A.4.2.7 XBefore transitivity . . . 150
A.4.2.8 Boolean operators mixed with XBefore . . . 151
A.4.3 Formulas as ATF . . . 155
A.4.3.1 Basic properties of ATF . . . 155
A.4.3.2 Associativity of ATF . . . 157
A.4.3.3 Equivalences in ATF . . . 158
A.4.3.4 Transitivity in ATF . . . 159
A.4.3.5 Mixed operators in ATF . . . 160
A.4.4 Equivalence of the new definition of XBefore with the old one . . . 161
A.4.5 Soundness and completeness on the mapping rules . . . 161

1 Introduction

The development process of critical control systems is based essentially on the rigorous execution of guides and regulations [4, 5, 6, 7]. Specialized agencies (like FAA, EASA and ANAC in the aviation field) use these guides and regulations to certify such systems.

Safety is a system's attribute that plays a crucial concern on critical systems and it is the responsibility of the safety assessment process. To employ such a process, dependable systems taxonomy and safety assessment techniques must be well defined and understood. Clarification of concepts of dependable systems can be surprisingly difficult when systems are complex, because the determination of possible causes or consequences of failure can be a very subtle process [8].

ARP-4761 [7] defines several techniques to perform safety assessment. One of them is Fault Tree Analysis (FTA). It is a deductive method that uses trees to model faults and their dependencies and propagation. In such trees, the premises are the leaves (basic events) and the conclusions are the roots (top events). Intermediary events use gates to combine basic events and each kind of gate has its own combination semantics definition. Fault trees (FTs) that use only $\vee$ (OR) and $\wedge$ (AND) gates are called *coherent fault trees* [9, 10, 11, 12, 13]. They combine the events as *at least one shall occur* and *all shall occur*, respectively. To analyse FTs, their structures are abstracted as Boolean expressions called *structure expressions*. The analysis of coherent FTs uses a well-defined algorithm based on the Shannon's method to obtain minimal cut sets (MCSs) from the structure expressions, and a general formula to calculate the probability of top events. The MCSs are obtained by reducing structure expressions to a normal form, in which each term is a combination of variables (basic events) with conjuctive (AND) gates, and the terms are combined as disjunctive (OR) gates. These minimal terms are also called *prime implicants* or *minterms*. The Shannon's method originated a formalism to reduce structure expressions called Binary Decision Diagram (BDD) [14, 15]. Another approach to reduce structure expressions is to use a mathematical model—called Free Boolean Algebra (FBA) [16, pp. 256-266]—that uses sets of sets to represent Boolean expressions.

Although structure expressions are formulas with logical operators, they are formalisms to enable automatic FTA. As shown in [17], FTs are a much richer model enabling a visual indication of fault paths, and includes description of subsystems as intermediate events.

Figure 1 shows how FTA is traditionally performed. It starts with an architectural model, then faults are identified and modelled in an FT. System requirements are identified and are checked with FTA results. If the requirements are satisfied (accepted), the process ends and the modelled system may be implemented. Otherwise, fault tolerance patterns are used, adding or modifying the original architecture to improve dependability. The analyses are executed until system requirements are met.

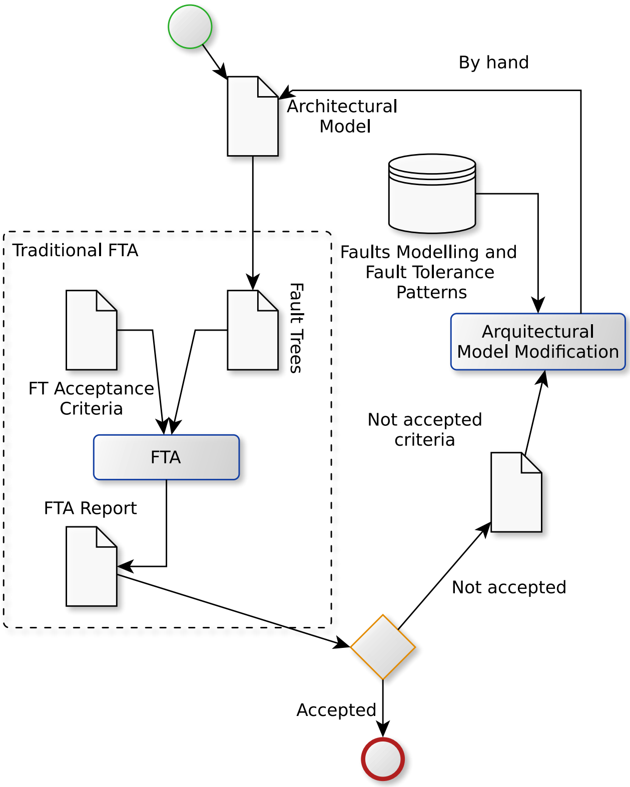

By hand
Architectural Model
Faults Modelling and Fault Tolerance Patterns
Arquitectural Model Modification
Traditional FTA
Fault Trees
FT Acceptance Criteria
FTA
FTA Report
Not accepted criteria
Not accepted
Accepted

Figure 1 – Traditional FTA

Besides the traditional OR and AND gates, the Fault Tree Handbook [18] defines other gates. For example the priority-AND (PAND) gate, which considers the order of occurrence of events. Although the Fault Tree Handbook defines these new gates, there is no algorithm to perform the analysis of trees that contain such new gates. This and the need of analysis of dynamic aspects of increasingly complex systems motivated the introduction of two new kinds of fault trees: Dynamic Fault Trees (DFTs) [1, 2] and Temporal Fault Trees (TFTs) [19, 20, 21]. These variant trees can capture sequential dependencies of fault events in a system. The difference from TFT to DFT is that TFTs use temporal gates directly, while DFT does not—DFTs gates are an abstraction of temporal gates. To differentiate the fault trees as defined in the Fault Tree Handbook from the other two, we will call them Static Fault Trees (SFTs).

The work reported in [20] aims at performing the full implementation of the Fault Tree Handbook, adding temporal gates to its Pandora[1] methodology. It was this implementation that introduced the new concept of TFTs, cited previously. In such trees, events ordering is well-defined and an algebraic framework was proposed to reduce structure expressions to obtain minimal cut sequences (MCSeqs) and perform probabilistic analysis. Reducing expressions is also desirable to check for tautologies, for example.

---

[1] Pandora stands for: P-AND-ORA, which translates to Priority AND, Time.

DFTs introduce very different gates to capture dynamic configurations of systems: cold spare (CSp), functional dependency (FDEP), and sequence enforcing (SEQ). The semantics of the first is to add "backup" events, so the gate is active if the primary event and all spares are active. The second adds basic events dependency from a trigger event. The third forces the occurrence of events in a particular order. There is also a warm spare (WSp) gate that is slightly different from the CSp gate. They differ on the nature of sparing, whether fast (warm, always-on) or slow (cold, stand-by). The readiness of the backup system in a WSp gate is higher than in a CSp gate. The work reported in [22] shows an algebraic framework to compositionally reduce DFT gates to order-based gates and perform probabilistic analysis of structure expressions. Thus, despite some limitations for spare gates [23], the structure expressions used in TFTs and DFTs can be formulated in terms of a generic order-based operator.

The $\neg$ (NOT) operator is absent in the algebras reported in [20, 21, 3, 24]. There is no consensus about the relevance of its use: (i) it can be misleading, generating non-coherent analysis [11], or (ii) it can be essential in practical use [9]. Our concern is that the decision of the relevance of its use should not be due to the choice of events-occurrence representation. The algebra created in this work defines the NOT operator and allows its use, as we show in Chapter 4.

Hierarchically Performed Hazard Origin and Propagation Studies[2] (HiP-HOPS) [25] is a set of methods and tools to analyse FTs. The semi-automatic generation of FTs has architectural models and failure expressions as inputs. The failure expressions are in fact structure expressions of components or subsystems. These expressions are annotated in components and subsystems and describe how they fail. The tool combines these expressions with regard to the architecture to generate FTs. The work reported in [19] shows a strategy to use the semi-automatic FT generation of HiP-HOPS with Pandora to generate structure expressions of TFTs.

Architecture Analysis and Design Language (AADL) [26] is a standard language to model (among other features) system structure and component interaction. AADL has several tools to perform different analyses to obtain SFT to perform FTA. But AADLs assertions framework does not express order explicitly as needed for TFT and DFT analyses.

In previous work [27, 28], we proposed a systematic hardware-based faults identification strategy to obtain failure expressions as defined in HiP-HOPS for SFTs. We considered faults in components or subsystems, but if we obtain failure expressions of a whole system, they are in fact structure expressions of an FT. Our strategy throws away the ordering information of the fault event sequences to generate failure expressions for components or subsystems for SFTs. We focused on hardware faults because we assume that software does not fail as a function of time (wear, corrosion, etc). We inherited this view from our industrial partner, which assumes that functional behaviour is completely analysed by functional verification [29]. We followed industry common practices using Simulink diagrams [30] as a starting point. The work reported in [28] was based on Communicating Sequential Processes[3] (CSP_M) to allow an automatic analysis using the model checker FDR. Thus, our strategy required the translation from Simulink to CSP_M [31]. It then runs FDR to obtain several counter-examples (which are fault traces) ending in failures. For two case studies provided by our industrial partner we showed that our automatically created failure expressions match with the engineer's provided ones or are better (a weaker proposition).

---

[2] <http://www.hip-hops.eu/>

1.1 Research questions

Both TFT and DFT lack a first-order logic mathematical model like the one defined for SFT. For SFTs, mathematical models to reduce structure expressions are either based on set inclusion, with FBA, or through tree search, with BDD. Both are efficient. One important concern on employing FTA is whether an FT indeed represents a system behaviour. The work reported in [32] exposes this concern for DFTs, and the HiP-HOPS framework—related to SFTs and TFTs—aims at getting this issue sorted out. Our contribution to this issue for SFT is shown in [28, 27].

The mathematical model for TFT has a discontinuity between two activation states: (i) non-occurrence, and (ii) occurrence some time later. Such a discontinuity has some drawbacks as, for example, the impossibility to use NOT gates, and handling the specific case of non-occurrence with zeros in Temporal Truth Tables (TTTs). The reduction of structure expressions in TFT is based on a combination of: (i) algebraic reduction—which can unfortunately result in an infinite application of rules—, (ii) modularisation of independent subtrees (subtrees not always are independent), and (iii) dependency tree (DT) [33]—which are limited to seven basic events, due to exponential growth.

Most mathematical models [34, 35, 36] for DFT are based on the formalisation of discrete-time Markov chain (DTMC) [37, 17] or continuous-time Markov chain (CTMC) [38, 39] because DFTs were initially conceived to be a visual representation of such models. As both DTMC and CTMC are state-based, they experience the state-space explosion problem. The works reported in [40, 41, 7] show techniques to overcome this problem, but the reduction can be infeasible because it depends on systems' models particularly, whether they are independent or not.

---

[3] This variant "M" is the machine-readable version of CSP.

There are other approaches, however. For instance, a modified version of BDD to tackle events ordering, called Sequential Binary Decision Diagram (SBDD) [42, 43], to reduce structure expressions, and the work reported in [36], which proposes a conversion of DFT into dynamic bayesian network (DBN) [44] to perform probabilistic analysis.

The approach to tackle events ordering with SBDD [43] has two kinds of nodes: terminals and non-terminals (terminals are nodes with basic events, and non-terminals are nodes with two events and an operator). Although demonstrated in [45] that these unconventional nodes (non-terminals) generate correct and efficient Boolean analysis, the analysis is still dependent on the order-related operators because the relation of terminals and non-terminals is not established directly (non-terminals are seen as an independent node in [43]). For example, the occurrence of $A \rightarrow B$ is related to the occurrence of A and B, but this relation is obtained in a further step, not in the SBDD.

The approach using the construction of DBNs [36] is automatic and handles time slices as $t + \Delta t$, which implies a notion of events ordering as well. As it is focused in probabilistic analysis, qualitative analysis is not directly supported.

The works reported in [3, 43] show that DFT's operators can be converted into order-related operators, simplifying DFT analysis. Although the mathematical model presented in [3] establishes a denotational semantics for order-related operators, it lacks a formal method for expression reduction based on such a model. It defines, instead, several algebraic laws to reduce expressions and an algorithm to minimize the structure function.

HiP-HOPS proposes a hierarchical approach to model systems and perform FTA (and Failure Modes and Effects Analysis (FMEA) [46]). Although there is a tool to model and analyse systems using HiP-HOPS, FTs construction is based on an algorithm, without proofs for soundness or completeness.

Another concern, left untreated in the literature, is the undesirable possibility of non-determinism in structure expressions. For example, if a commission is observed when fault A is active and an omission is observed when faults A and B are active, then the system may behave non-deterministically with a commission or omission when both A and B are active (A and B implies A).

From the exposed in this section, our research questions are:

RQ_1) Is there a consistent mathematical model to analyse TFTs and DFTs that is set-based and similar to FBA?

RQ_2) What guarantees can we provide to detect non-determinism in erroneous behaviour?

Also, does such a model:

- RQ_3) have the capacity of representing events ordering similar to TFT and DFT?
- RQ_4) represent systems behaviour by construction?
- RQ_5) allow both qualitative and quantitative analyses as supported by TFT and DFT?

1.2 Proposed solution

In this work we present an algebra, called Algebra of Temporal Faults (ATF), to express ordering of fault events (TFT and DFT), enabling analysis of acceptance criteria of FTs. The laws of ATF are given in a denotational semantics based on sets of lists of distinct elements. ATF aims at answering the research questions RQ_1 and RQ_3. The analysis of acceptance criteria is a decision problem and we use first-order logic and Isabelle/HOL 2015[4] as verification tool. Indeed, ATF is part of a bigger strategy that relates fault injection on nominal models, fault modelling, FTA, and fault tolerance patterns.

System and fault modelling is an essential step towards safety analysis. Architectural modelling is the first step of the strategy and can be executed either in a graphical tool, or as requirements in natural language. For example, our work reported in [48, 49] uses fault modelling in Systems Modelling Language (SysML) [50] to verify fault tolerance of Systems of Systems (SoSs) [51].

Writing and analysing expressions with order-related operators is more complex than analysing expressions with Boolean operators only. The Activation Logic (AL) works together with an inner algebra to perform analysis of system structure and component interaction with a focus on fault modelling and fault propagation, tackling the complexity introduced by order-related operators. AL receives an algebra and the set of operational modes of a system as parameters. The choice of algebra defines which structure expressions can be obtained: if Boolean algebra is passed as a parameter, the AL can generate structure expressions with Boolean operators (SFT); if the ATF is passed as a parameter, the AL can generate structure expressions with order-related operators (TFT and DFT). The AL requires that the inner algebras provide a set of properties (tautology and contradiction) and semantic values. The use of the NOT is essential: besides its use in expressions, we use the complement to normalise the expressions to provide *healthy* expressions.

To obtain critical event expressions used in FTs and to denote faults propagation, the AL provides a predicates notation and verification of non-determinism. We show three different approaches to check the non-determinism and answer research question RQ_2: (i) verify its existence, (ii) indicate which set of operational modes are active for a combination of faults, or (iii) what is the combination of faults that activates a set of operational modes.

---

[4] The 2002 tutorial is reported in [47], but there is a newer version published with the tool itself. The tool and the tutorial are available on their website at <http://isabelle.in.tum.de>.

In our proposed solution, depending on how much clear are the faults, the analyst may follow one of the paths: (i) model the system in Simulink to allow fault injection and discovery, or (ii) model faults using the Activation Logic. Both paths ends with structure expressions and the FTA is performed using ATF.

Figure 2 shows how to perform FTA using faults injection. The "Faults injection" block is obtained from part of our work reported in [28, 27]. It starts with Simulink modelling, converts the model to CSP_M and then obtains fault event sequences. The fault event sequences are then mapped to ATF, which have a denotational semantics based on sets of lists. This strategy aims at answering the research question RQ_4.

Safety requirements are stated in terms of critical failures as, for example, "the probability of a complete failure of an airplane engine should be less than 10^{-9}" (quantitative), or "a complete failure of the propulsion system shouldn't be caused by a single failure" (qualitative). In this work we call *acceptance criteria* the verification of a safety requirement on an FT, where FT's top event is the undesired failure. Positive requirements as, for example, "the communications system should be operational 99.99% of the cruise phase" are treated as a complement (the complete failure should have a probability in less than 0.01% of the cruise phase). The acceptance criteria analysis aims at answering the RQ_5.

From the model in ATF (Figure 2), the acceptance criteria are then verified. If the criteria are accepted the process finishes. Otherwise the system is modified, and the process continues, modifying system's architecture, using fault tolerance patterns, improving system's dependability.

Figure 3 shows a faults modelling strategy directly in the AL. The logic associates each operational mode with a faults expression. After modelling all faults, the top events are extracted in a predicates notation. For example, "does the behaviour of the system is the operational mode X?", where X can omission, commission, etc. Given the flexibility of the AL notation it is used to reason about basic fault events and top-event failures, which are related to RQ_3. Each predicate in AL generates an expression in ATF, which are reduced to obtain a canonical form to obtain MCSeqs and to calculate top-events probability. With the system modelled in AL, the fault tolerance patterns can be applied directly on the model.

Joining the paths described in Figure 2 and Figure 3, the complete proposed solution is summarized in Figure 4 (A and B, respectively).

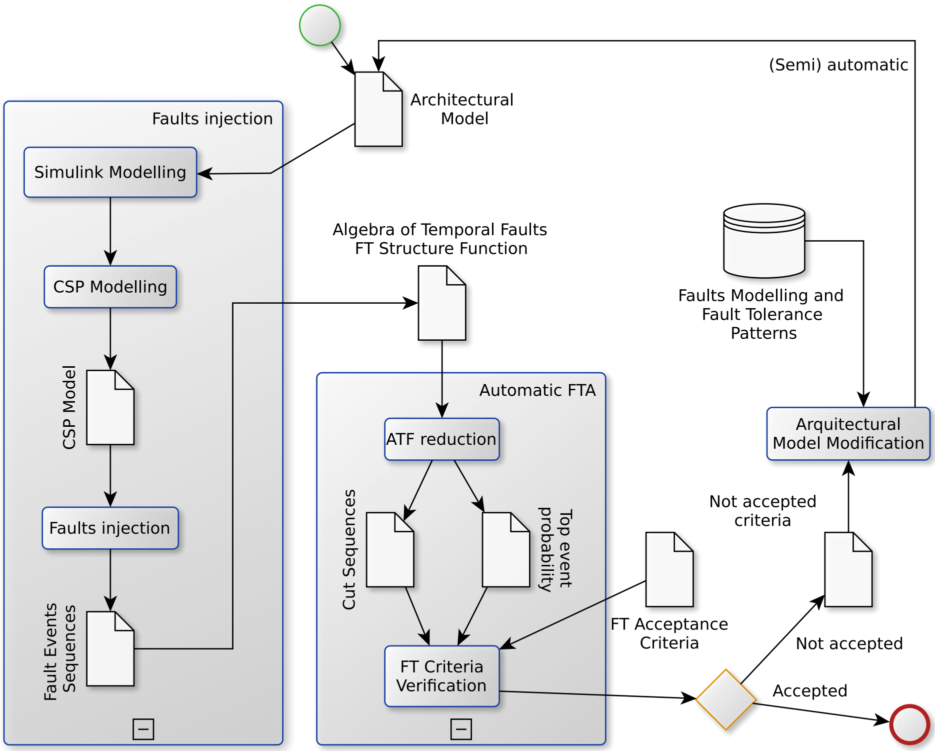

(Semi) automatic
Architectural Model
Faults injection
Simulink Modelling
CSP Modelling
CSP Model
Faults injection
Fault Events Sequences
Algebra of Temporal Faults
FT Structure Function
Automatic FTA
ATF reduction
Cut Sequences
Top event probability
FT Criteria Verification
FT Acceptance Criteria
Faults Modelling and Fault Tolerance Patterns
Arquitectural Model Modification
Not accepted criteria
Not accepted
Accepted

Figure 2 – Faults injection and ATF to perform FTA

1.3 Contributions

The main contributions of this work are:

C_1) Define a denotational and algebraic model to express fault events order with ATF (Chapter 4);

C_2) Define a new operator to express order explicitly and proving that the resulting algebra—(ATF) using this operator and Boolean operators—is a conservative extension of the Boolean algebra (also published in [52])—see Chapter 4;

C_3) From Simulink models, obtain fault event sequences and mapping to ATF (Chapter 4);

C_4) Reason about fault modelling in AL, to obtain formal expressions of critical failures (top-event failures, Chapter 5);

C_5) Illustrate the application of the laws on a real case study, provided by our industrial partner (Chapter 6).

We use Isabelle/HOL, theories in Isabelle/HOL's library, and a theory in the AFP library [53] to prove all theorems presented in this work.

The contributions cover the following scenarios, presented as a case study in Chapter 6:

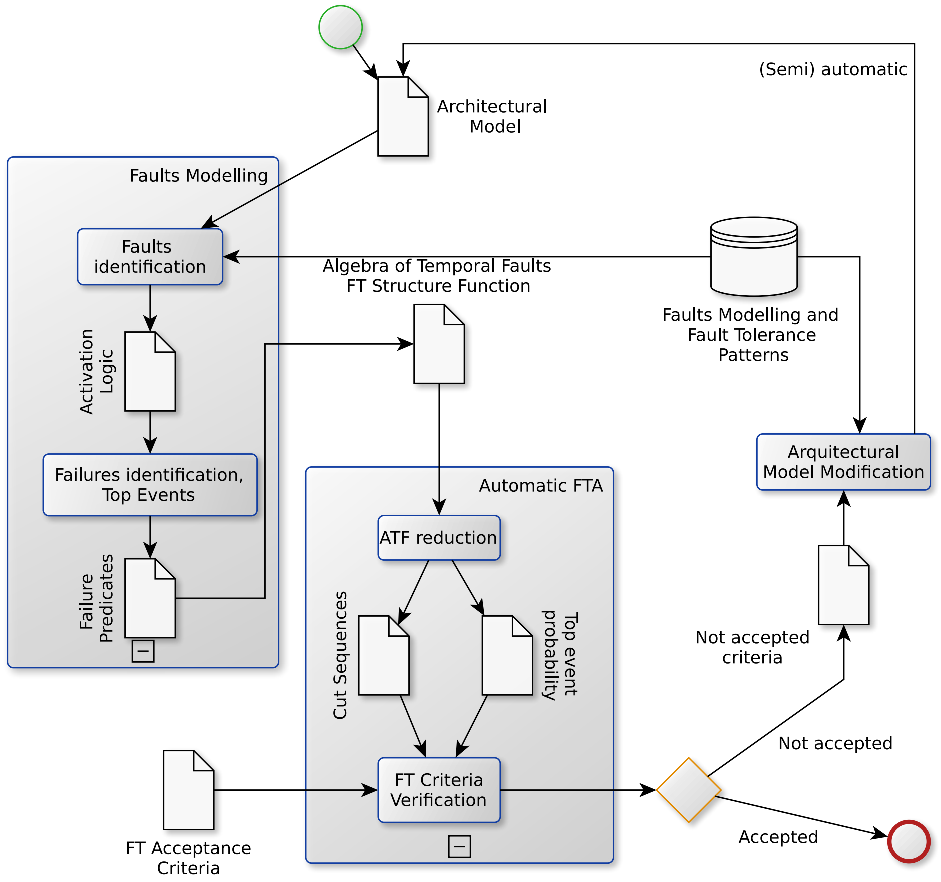

(Semi) automatic
Architectural Model
Faults Modelling
Faults identification
Algebra of Temporal Faults FT Structure Function
Faults Modelling and Fault Tolerance Patterns
Activation Logic
Arquitectural Model Modification
Failures identification, Top Events
Automatic FTA
ATF reduction
Failure Predicates
Cut Sequences
Top event probability
Not accepted criteria
Not accepted
FT Criteria Verification
FT Acceptance Criteria
Accepted

Figure 3 – AL and ATF to perform FTA

1. From a model in Simulink, obtain the failure expression of a critical failure, analyse the ordering relation of fault events, and verify its acceptance criteria;
2. Given a set of FT structure expressions, verify which fault combinations analysis are missing.

1.4 Thesis organization

This thesis is organized as follows: in Chapters 2 and 3 we show the concepts and tools used as basis for this work. Chapter 4 presents ATF, Chapter 5 presents AL, Chapter 6 the case study and the application of the proposed strategy, and we present our conclusions and future work in Chapter 7. The contributions presented in this work are summarized in terms of proved results. To facilitate the understanding of the presented strategy, the effort to build laws and theirs (mechanized) proofs are shown in Appendix A.

Isabelle/HOL's theory files with all proofs are available at <http://www.cin.ufpe.br/~alrd/phd/phd-alrd.zip> (password: 6Zvq$5Vyj).

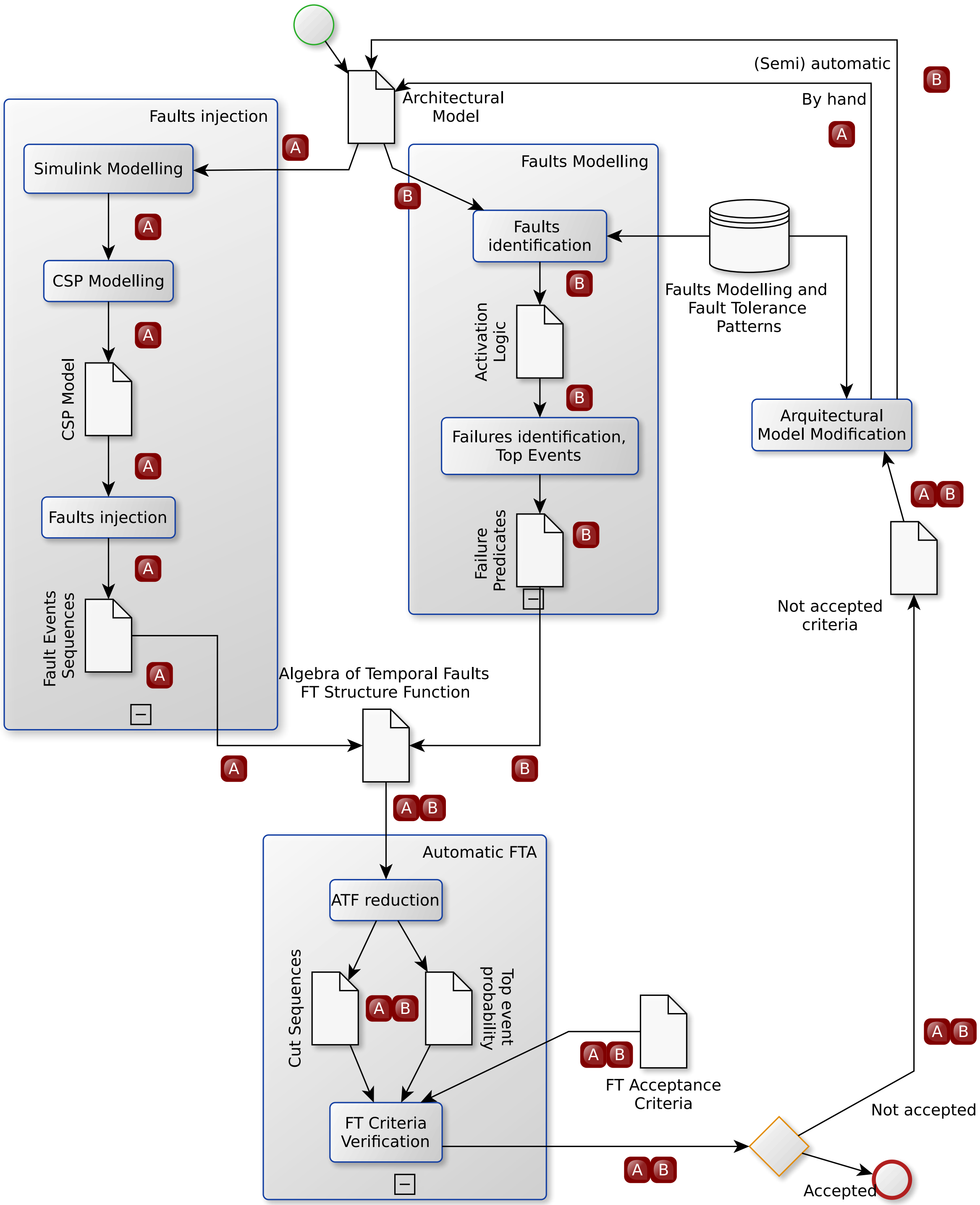

(Semi) automatic
B
Architectural
Model
By hand
A
Faults injection
A
Simulink Modelling
A
CSP Modelling
A
CSP Model
A
Faults injection
A
Fault Events
Sequences
A
Faults Modelling
B
Faults
identification
Faults Modelling and
Fault Tolerance
Patterns
B
Activation
Logic
B
Failures identification,
Top Events
Arquitectural
Model Modification
A B
Failure
Predicates
B
Not accepted
criteria
Algebra of Temporal Faults
FT Structure Function
A
B
A B
Automatic FTA
ATF reduction
Cut Sequences
A B
Top event
probability
A B
FT Acceptance
Criteria
A B
Not accepted
FT Criteria
Verification
A B
Accepted

Figure 4 – Strategy overview

2 Basic concepts

Means to dependability are obtained by modelling and analysing a system. It is strongly related to fault modelling, which depends on the kinds of analyses we want to perform. FTs are present in several stages of systems' modelling. We introduce dependability and fault modelling in Section 2.1.

An SFT is a snapshot[1] of a faults topology of a system, subsystem or component. The time relation of fault events in TFTs and DFTs allows the analysis of different configurations (or snapshots) of a system, subsystem or component. We discuss these time relations in Section 2.2.

2.1 Systems, dependability, and fault modelling

Computing systems are characterized by five properties: functionality, performance, cost, dependability, security. The work reported in [54, p. 289–302] explain these properties—including dependability—with a focus on software. Hardware and software are connected, as software faults may cause a failure in a software-controlled hardware, and hardware faults may send incorrect data, causing a failure in the software.

The work reported in [8] summarizes all concepts of and related to dependability for computing systems that contain software and hardware. In the following, we show these concepts and highlight those used in this work.

2.1.1 Systems

Before introducing systems' dependability, we first describe what a system is and its characteristics. A *system* is an entity that interacts with other systems (software and hardware as well), users (humans), and the physical world. These other entities are the *environment* of the given system, and its *boundary* is the frontier between the system and its environment.

The *function* of a system is what the system is intended to do, and its *behaviour* is what the system does to implement its function. The *total state* of a system are the means to implement its function and is defined as the set of the following states: computation, communication, stored information, interconnection, and physical condition. The *service* delivered by a system is its behaviour as it is perceived by its boundary. A system can both provide and consume services.

---

[1] Whether a top event indeed causes a catastrophic or major failure is out of the scope of this thesis; we consider that, if it is possible that such failure occur, then it will.

The *structure* of a system is how it is composed: a system is composed of components, and each component is another system, etc. This concept of hierarchical compositionality in systems is what originated the concept of SoS and is the object of analysis in HiP-HOPS. Such a recursion (of a system containing other systems) stops when a component—or a constituent system—is considered to be atomic. A system is the total state of its atomic components.

2.1.2 Dependability

The concepts that create the basis for dependability are: (i) threats to, (ii) attributes of, and (iii) means to attain.

Threats to dependability are the so-called *fault-error-failure* chain. A failure is a service deviation perceived on systems' boundary. An error is the part of the total state of a system that leads to subsequent service failure. Depending on how a system tolerate internal errors, many errors may not reach system's boundary. Finally, a fault is what causes an error. In this case, we say that the fault *occurred* (the fault is active). Otherwise, the fault is dormant, and has not occurred (yet). A *degraded* mode of a system is when there are active faults, so some functions of the system are inoperative, but the system still delivers its service.

There are two acceptable definitions of dependability reported in [8]. One is more general, difficult to measure: "the ability to deliver service that can justifiably be *trusted*". A more precise definition that uses the definition of service failure is: "the ability to avoid service failures that are more frequent and more severe than is acceptable". This definition has two implications about system's requirements: there should be defined how it can fail, and what are the acceptable severity and frequency of its failures.

The following systems' dependability attributes enlightens such requirements:

Availability: the readiness for correct service;

Reliability: continuity of correct service;

Safety: absence of catastrophic consequences on the environment (other systems, users, and the physical world). Safety can be verified using FTs, which is part of the objective of this work;

Integrity: absence of improper systems alterations;

Maintainability: ability to be modified and repaired.

A system description should mention all or most of these attributes, at least the first three of them.

The implementation of these attributes requires a deep analysis of system's models. The *means to attain dependability* are summarized as follows:

Prevention is about avoiding incorporating faults during development.

Tolerance deals with usage of mechanisms to still deliver a—possibly degraded—service even in the presence of faults.

Removal is about detecting and removing (or reducing severity of) failures from a system, both in the development and production stages.

Forecasting is about predicting likely faults so they can removed, or tackling their effects.

The intersection of the current work with dependability is in fault removal during development and fault tolerance (analysis). Following the taxonomy presented in [8], there are some techniques for fault removal, summarized as follows:

a) Static verification:
   - Structural model:

     Static analysis: Range from inspection or walk-through, data flow analysis, complexity analysis, abstract interpretation, compiler checks, vulnerability search, etc.

     Theorem proving: Check properties of infinite state models.
   - Behaviour model:

     Model checking: Usually the model is a finite state-transition model (Petri-nets (PNs), finite state automata). Model-checking verifies all possible states on a given system's model.

b) Dynamic verification:
   - Symbolic inputs:

     Symbolic Execution: It is the execution with respect to variables (symbols) as inputs.
   - Actual inputs:

     Testing: Selected input values are set on system's inputs and their outputs are compared to expected values. The verification outcomes are observed faults, in case of hardware testing or software mutation testing, and criteria-based, in case of software testing.

Verification methods are often used in combination. For example, symbolic execution may be used to obtain testing patterns, test inputs can be obtained by model-checking as in [55], faults can be used as symbolic inputs, and system behaviour can be observed using model-checking as in [28, 27] (This technique is called fault injection; see also [56]).

The techniques to attain fault tolerance are summarized as follows:

Error detection: is used to identify the presence of an error. It can be a concurrent or a preemptive detection. Concurrent detection takes place during normal service, while preemptive detection takes place while normal service is suspended.

Recovery: transforms a system state that contains errors into a state without them. The behaviour of the system upon recovery is equivalent to the normal behaviour. Techniques range from rollback to a previously saved state without errors, error masking, isolation of faulty components, to reconfiguration using spare components.

In this work, we use a combination of: (i) fault-injection, (ii) theorem proving, and (iii) symbolic execution. We use these methods to obtain an erroneous behaviour of the system which is compared to the system dependability attributes (safety). We explain how these methods are combined in Chapter 4.

On the analyses of systems and its constituents, there is a distinction of operational modes and error events. Operational modes refer to the behaviour that is perceived on the boundaries of a system (*failure*). Error events, on the other hand, represent the behaviour detected in a constituent of a system. Such error events may relate to an operational mode, but not necessarily. Further in Chapter 4 we abstract these differences and leave the distinction as a parameter. We refer to such a set as *operational modes*.

2.1.3 Fault Modelling

Fault modelling plays an important role in reasoning about the fault-error-failure chain. They are the initial steps to perform the verification of a system, starting in the architectural model to reason about the critical failures, which are (in general) the top-events in FTs.

SysML is a profile for Unified Modelling Language (UML) that provides features to model structure and behaviour of systems. The works reported in [48, 49] define several structural and behavioural views in SysML to model the fault-error-failure chain and fault tolerance. Fault, error, failures, and fault propagation have structural views, which are related to behavioural views to describe fault activation and recovery. These works map SysML to two formal languages—COMPASS Modelling Language (CML) [57] and Communicating Sequential Processes (CSP) [58], respectively—to verify fault tolerance.

In [7] the safety assessment process for civil airborne systems and equipment describes development cycles and methods to “clearly identify each failure condition”. The methods that involve failure identification are: (i) SFT, (ii) Dependence Diagram[2] (DD) [59, p. 198], (iii) Markov chain, and (iv) FMEA. The first three are top-down methods, that start with undesired failure conditions and moves to lower levels to obtain more detailed conditions that causes the top-level event. DDs are an alternative method of representing the data in SFT. FMEA is a bottom-up method that identifies failure modes of a component and determines the effects on the upper level. We detail SFT in Subsection 3.1.1.

DFTs are an extension of SFTs and models dynamic behaviour of system faults. Similarly to the relation of SFTs and DDs, the work reported in [60] demonstrates the relation of DFTs to Dynamic Reliability Block Diagrams (DRBDs) [60]. As the models (DFT and DRBD) are equivalent, this work sticks to DFT due to the amount of work already published. We detail DFTs in Subsection 3.1.3.

2.2 Time relation of fault events

The most general case for time relations is to consider that each fault event has a continuous time duration. They are the basis on how fault events discretisation are defined. The point of view in this work is the analysis of the effects caused by a combination of faults in a snapshot of a system state. In Figure 5 we show all possibilities of events relations in a continuous time line (from A to B; the converse relation is similar):

1. A starts before and ends after B has started, but before B has ended;
2. A starts before B and ends after B has ended (A contains B);
3. B starts after A, but they end at the same time;
4. A and B start at the same time, but A ends before B;
5. A and B start and end at the same time;
6. A starts before B and ends when B starts.
7. A starts and ends before B starts;

Considering that fault occurrence corresponds to the start of a fault event and its duration, from Figure 5 we clearly identify which event comes first: A comes before B, except in the cases (d) and (e), where they start exactly at the same time. If fault events are independent (they are not susceptible to have a common cause) then the probability of their occurrences starting at the same time is very low. The relations (f) and (g) shows the case that the system was repaired, thus A is not active when B starts. In **??** we abstract the relation of events in continuous time as an *exclusive before* relation, based on fault *occurrence* (it is similar—at least implicitly—to what is reported in [20, 22]).

---

[2] Also known as Reliability Block Diagram (RBD).

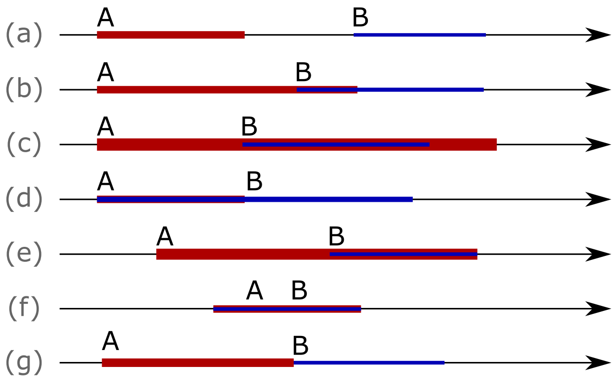

(a)
A
B
(b)
A
B
(c)
A
B
(d)
A
B
(e)
A
B
(f)
A
B
(g)
A
B

Figure 5 – Relation of two events with duration

3 Analysis and tools

Structure expressions are used to analyse fault trees. In general, a structure expression comes from gates semantics and basic events. Basic events become variables and gates become operators (a gate may become one or more operators). In Section 3.1 we explain SFTs, TFTs, DFTs, and their respective structure expressions.

FBAs and BDDs are the basis to analyse structure expressions. Also, we were inspired by FBA concepts to create our Algebra of Temporal Faults (Chapter 4). We explain BDDs and derived techniques in Section 3.2, and FBAs in Section 3.3.

The use of the Boolean operator NOT: (i) can be misleading, generating non–coherent fault trees, or (ii) can be essential in practical use. We discuss such cases in Section 3.4.

To reuse a nominal model to analyse faults we need fault injection. In Section 3.5 we explain how we used Simulink and CSP_M to inject faults and obtain failure expressions from a nominal model.

Finally, in Section 3.6 we present basic usage of Isabelle/HOL and Intelligible semi-automated reasoning (Isar), which were essential to carry out the proofs presented in this thesis.

3.1 Fault Tree Analysis and structure expressions

FTA was introduced in the Fault Tree Handbook [18] with Static Fault Trees. FTA is a deductive method that investigates what are the possible causes of an unwanted event. The method starts with the top-level event as the unwanted event and the combination of lower-level events that can cause it. Events are combined using gates, and each gate has a well defined semantics. It continues until basic (atomic) events are reached. An SFT represents, in a single view—very often considering faults outside of the boundaries of a system—different states in which a particular failure (top event) is active in a system. The most traditional gates are AND and OR, which are equivalent to Boolean operators. These gates are also called coherent gates because they construct coherent trees (see Section 3.4 about the use of NOT gates). The Fault Tree Handbook shows other gates as, for example, the PAND gate, but the FTA with these gates is not well defined. SFT's gates and analysis are detailed in Subsection 3.1.1.

TFTs were created aiming at fully implementing the Fault Tree Handbook. The PAND gate was first defined for SFTs, but its analysis was left open in the handbook. The semantics (and analysis) of TFTs is defined in terms of a denotational semantics based on *sequence values* to express ordering of events, thus tackling PAND's order. We explain TFTs and the sequence values in Subsection 3.1.2.

With component and system design evolution, DFTs were created to tackle dynamic behaviour: fault-tolerance-related components (CSp), functional dependency (FDEP), and analysis of particular order of occurrence of faults (SEQ). SFT's gates are still present as DFT's gates. We explain them and DFT's analysis in Subsection 3.1.3.

The structure of an FT (or the structure of an MCS, explained further) is represented with a formula. The variables represent occurrences of basic events. Unary and binary relation symbols capture the semantics of gates. A formula with these characteristics is called *structure expression* or *structure function* (as the expression depends on the variables). The semantics of a structure expression is that the top-level event occurs if some combination of basic events occur.

The results obtained from the FTAs are shown in the Fault Tree Handbook. We summarize them as:

a) Qualitative

MCSs: Smallest combinations of component failures causing system failure. They are obtained from the reduction of structure expressions to a normal form. For example, in SFTs, structure expressions are reduced to disjunctive normal form (DNF). Each term in a reduced DNF is an MCS.

Importance: Qualitative rankings on contributions to system failure. A single fault causing a catastrophic failure is usually unacceptable. Ranking MCSs is the same as ordering them in ascending order of their size (smaller first).

b) Quantitative

Numerical probabilities: Probabilities of system and MCS failures. A system failure probability is obtained by assigning probabilities to basic events and then calculating it according to the gate semantics. MCS failure probability is the calculation of the probability of the occurrence of *all* basic events of a specific MCS.

Importance: Quantitative rankings on contributions to system failure. Ranking MCSs is the same as ordering them in descending order of some unreliability formula (higher first). These formulas used to calculate importance vary. The most common are: (i) system unavailability, and (ii) system failure occurrence rate.

Sensitivity evaluation: Modifying characteristics of components and evaluate their impact. For a particular event in a tree, a higher and a lower failure probability value are assigned. If the system unavailability is not changed, then such an event is not important—the system is not sensitive to such an event.

As stated in [61], there are other uses of FTA. One of great importance is using it to minimize and optimize resources, which has been object of study in HiP-HOPS [62]. Through importance measures, FTA not only identifies what is important but also what is unimportant. This removes components without impacting the overall failure probability, which is related to the quantitative importance and sensitivity evaluation.

In important stages of critical systems, FTA plays an essential role. At least three dependability means can be achieved using FTs:

Removal. An FTA calculates the probability of failure of a subsystem. If such a probability is higher than a certain maximum reference, such a subsystem should be removed or left to be incorporated in combination with a more reliable component.

Tolerance. An FTA indicates whether a single fault—or fewer combinations than expected—could lead to a catastrophic failure. In this case, a system should have replication, or stages of fault detection and error handling. Also, the probability of failure of the chosen fault tolerance method can be evaluated.

In Subsections 3.1.1 to 3.1.3 we briefly show FT symbology and means to analyse FTs. We detail its structure expression extraction because they are a common means to perform both qualitative and quantitative analysis.

3.1.1 Static Fault Trees

SFT gates and structure expressions were used as basis for other kinds of tree, as in TFTs and DFTs. We explain their symbology and semantics in this subsection.

The Fault Tree Handbook shows traditional symbols for gates and events. Basic events are usually drawn as rectangle (for the text) and a circle below it, as shown in Figure 6, or as a circle with the text of the basic event, as shown in Figure 7. Top-level and intermediary events are drawn as a rectangle (for the text) and a gate below it, as shown in Figures 6 and 7. When an FT becomes too large, transfer in and out symbols can be used. They are usually drawn as triangles with a letter or a number. Figure 7 depicts traditional gates as specified in the Fault Tree Handbook, and Figure 6 shows an FT using the Fault Tree Analyser[1]—a free commercial tool. In this work, to keep a visual identity with other FTs, and to avoid symbol confusion, we use gate symbols as shown in Figure 8.

Structure expressions in FTA are defined in terms of set theory, using symbols for fault events occurrence. If a fault event symbol is in a set, then it means that this fault has

---

[1] <http://www.fault-tree-analysis-software.com>, accessed 2/feb/2016

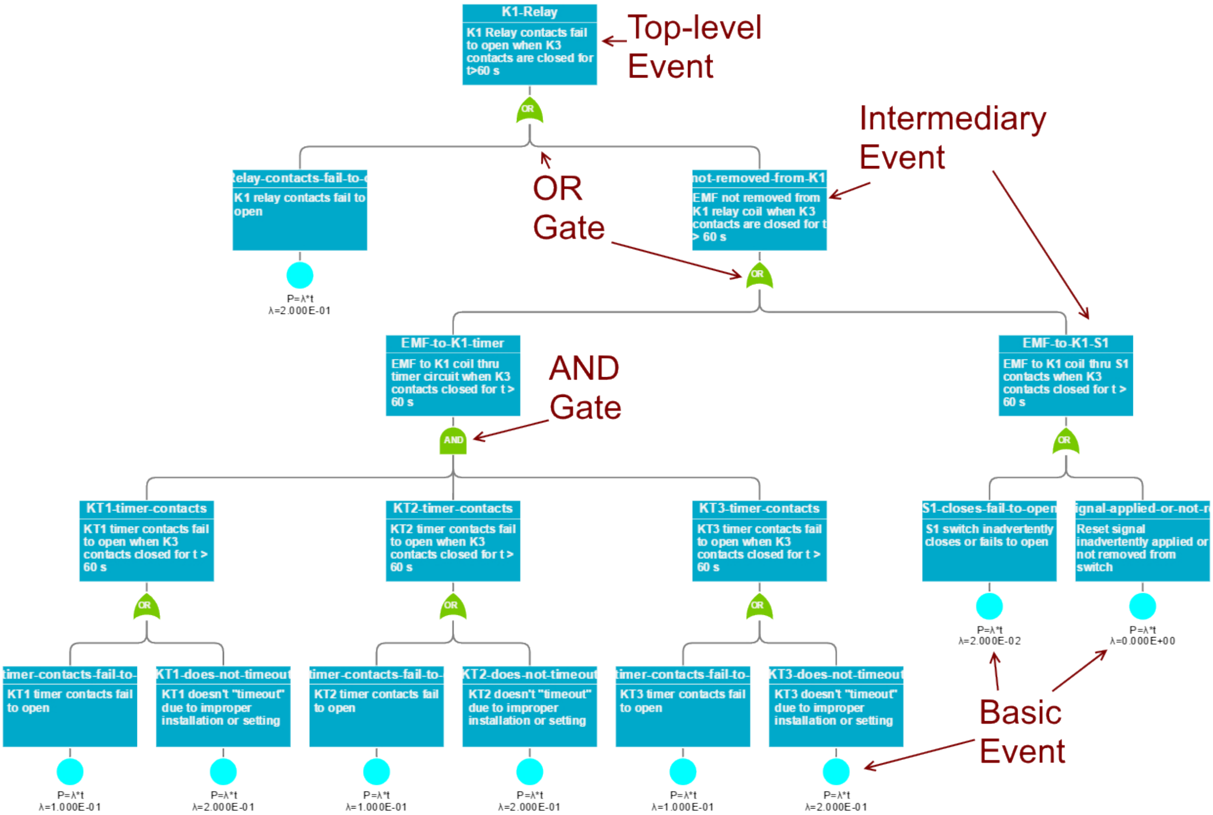

K1-Relay
K1 Relay contacts fail to open when K3 contacts are closed for t>60 s
Top-level Event
OR
Intermediary Event
elay-contacts-fail-to-
K1 relay contacts fail to open
P=λ*t
λ=2.000E-01
OR Gate
not-removed-from-K1
EMF not removed from K1 relay coil when K3 contacts are closed for t > 60 s
OR
EMF-to-K1-timer
EMF to K1 coil thru timer circuit when K3 contacts closed for t > 60 s
AND Gate
AND
EMF-to-K1-S1
EMF to K1 coil thru S1 contacts when K3 contacts closed for t > 60 s
OR
KT1-timer-contacts
KT1 timer contacts fail to open when K3 contacts closed for t > 60 s
OR
KT2-timer-contacts
KT2 timer contacts fail to open when K3 contacts closed for t > 60 s
OR
KT3-timer-contacts
KT3 timer contacts fail to open when K3 contacts closed for t > 60 s
OR
S1-closes-fail-to-open
S1 switch inadvertently closes or fails to open
P=λ*t
λ=2.000E-02
ignal-applied-or-not-r
Reset signal inadvertently applied or not removed from switch
P=λ*t
λ=0.000E+00
timer-contacts-fail-to-
KT1 timer contacts fail to open
P=λ*t
λ=1.000E-01
KT1-does-not-timeout
KT1 doesn't "timeout" due to improper installation or setting
P=λ*t
λ=2.000E-01
timer-contacts-fail-to-
KT2 timer contacts fail to open
P=λ*t
λ=1.000E-01
KT2-does-not-timeout
KT2 doesn't "timeout" due to improper installation or setting
P=λ*t
λ=2.000E-01
timer-contacts-fail-to-
KT3 timer contacts fail to open
P=λ*t
λ=1.000E-01
KT3-does-not-timeout
KT3 doesn't "timeout" due to improper installation or setting
P=λ*t
λ=2.000E-01
Basic Event

Figure 6 – SFT symbols using a free commercial tool

occurred. A set is a combination of fault events that causes the occurrence of the top-level event of a tree. A structure expression of a tree is denoted by a set of sets of fault event combinations. The OR gate becomes the union operator between sets and the AND gate, the intersection. For example, if a system contains fault events a, b, and c, fault trees for this system contain at most all these three events. The occurrence of the fault event a is denoted by a set of sets A, which contains the following sets:

a) $\{a\}$: only a occurs;

b) $\{a, b\}$: a and b occur in any order;

c) $\{a, c\}$: a and c occur in any order;

d) $\{a, b, c\}$: all three events occur in any order.

All sets of A contain the fault event a. Similarly, the sets of sets B—that represents the occurrence of b—contains all sets that contain the fault event b (it includes the set $\{a, b, c\}$, for example).

The fault tree in Figure 9 contains only two events and the resulting structure expression for this FT is the expression $A \cap B$ (TOP), where A and B are the sets of sets that contain a and b, respectively. The resulting combinations for TOP are $\{a, b\}$ and $\{a, b, c\}$ (fault events a and b occur in all possibilities).

After obtaining structure expressions, the next step is to reduce the expressions to

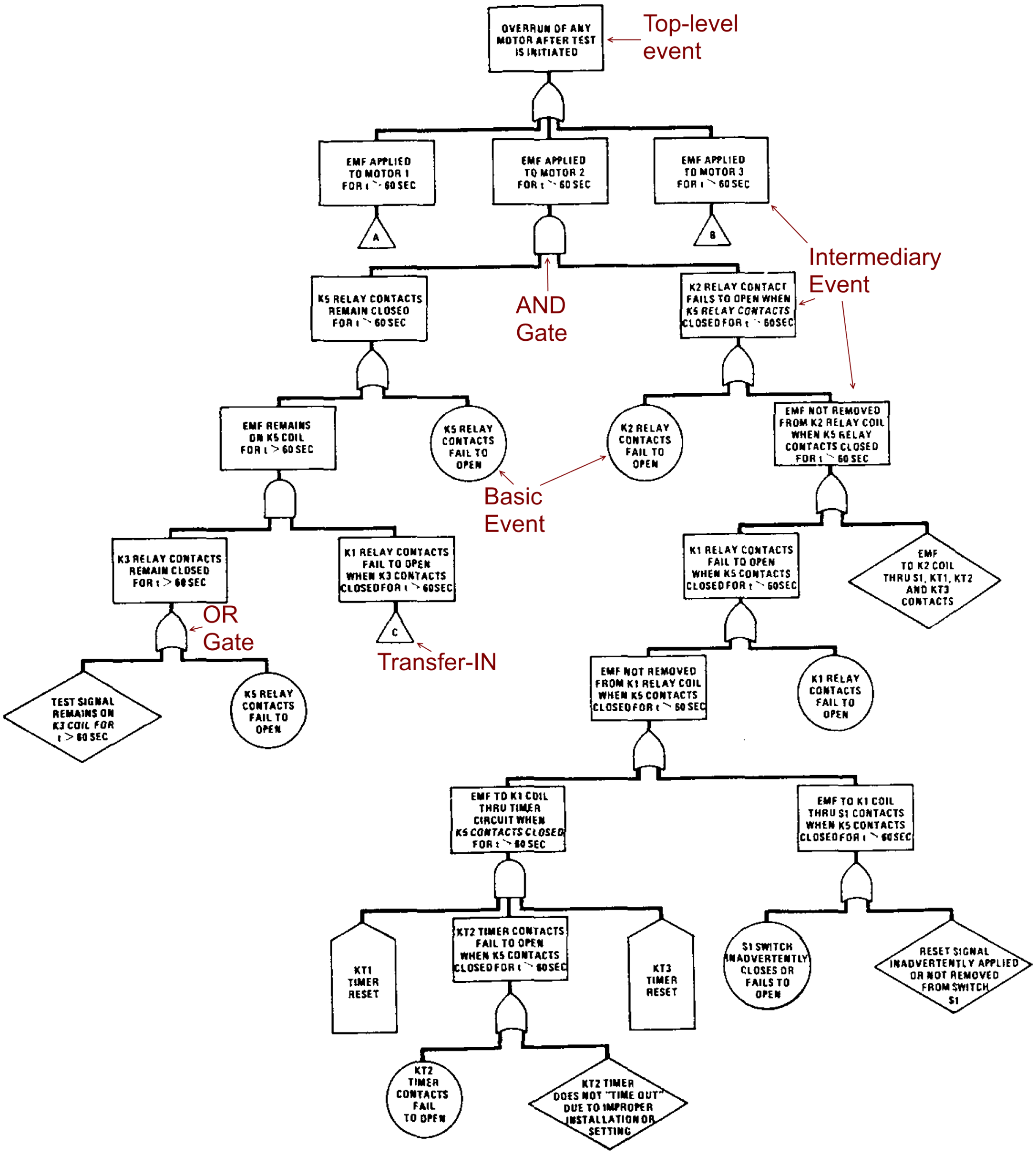
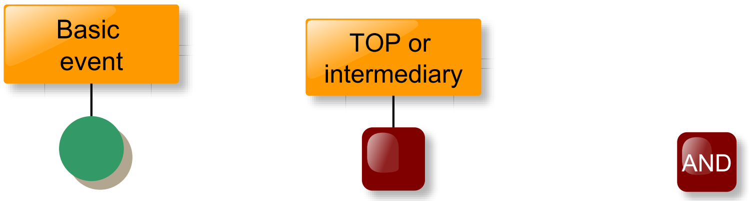

OVERRUN OF ANY MOTOR AFTER TEST IS INITIATED
Top-level event
EMF APPLIED TO MOTOR 1 FOR t > 60 SEC
EMF APPLIED TO MOTOR 2 FOR t > 60 SEC
EMF APPLIED TO MOTOR 3 FOR t > 60 SEC
A
B
Intermediary Event
AND Gate
K5 RELAY CONTACTS REMAIN CLOSED FOR t > 60 SEC
K2 RELAY CONTACT FAILS TO OPEN WHEN K5 RELAY CONTACTS CLOSED FOR t > 60SEC
EMF REMAINS ON K5 COIL FOR t > 60 SEC
K5 RELAY CONTACTS FAIL TO OPEN
K2 RELAY CONTACTS FAIL TO OPEN
EMF NOT REMOVED FROM K2 RELAY COIL WHEN K5 RELAY CONTACTS CLOSED FOR t > 60 SEC
Basic Event
K3 RELAY CONTACTS REMAIN CLOSED FOR t > 60 SEC
K1 RELAY CONTACTS FAIL TO OPEN WHEN K3 CONTACTS CLOSED FOR t > 60SEC
K1 RELAY CONTACTS FAIL TO OPEN WHEN K5 CONTACTS CLOSED FOR t > 60SEC
EMF TO K2 COIL THRU S1, KT1, KT2 AND KT3 CONTACTS
OR Gate
C
Transfer-IN
TEST SIGNAL REMAINS ON K3 COIL FOR t > 60 SEC
K5 RELAY CONTACTS FAIL TO OPEN
EMF NOT REMOVED FROM K1 RELAY COIL WHEN K5 CONTACTS CLOSED FOR t > 60 SEC
K1 RELAY CONTACTS FAIL TO OPEN
EMF TO K1 COIL THRU TIMER CIRCUIT WHEN K5 CONTACTS CLOSED FOR t > 60 SEC
EMF TO K1 COIL THRU S1 CONTACTS WHEN K5 CONTACTS CLOSED FOR t > 60SEC
KT1 TIMER RESET
KT2 TIMER CONTACTS FAIL TO OPEN WHEN K5 CONTACTS CLOSED FOR t > 60SEC
KT3 TIMER RESET
S1 SWITCH INADVERTENTLY CLOSES OR FAILS TO OPEN
RESET SIGNAL INADVERTENTLY APPLIED OR NOT REMOVED FROM SWITCH S1
KT2 TIMER CONTACTS FAIL TO OPEN
KT2 TIMER DOES NOT "TIME OUT" DUE TO IMPROPER INSTALLATION OR SETTING

Figure 7 – SFT symbols as in the Fault Tree Handbook

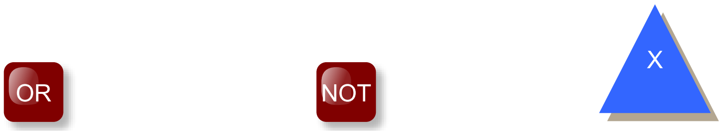
a canonical form to obtain the *MCSs*, which are the sets that contain the minimum and sufficient events to activate the top-level failure. Probabilistic analysis is then performed on these events to obtain the overall probability of occurrence of the top-level event. The Fault Tree Handbook shows an algorithm based on Shannon's method to reduce structure expressions to obtain minimal cut sets. The Boolean expression of the tree shown in Figure 9 is $TOP = A \wedge B$. A technique called BDD—which derives from Shannon's method—is explained in Subsection 3.2.2.

3.1.2 Temporal Fault Trees

There are at least two versions of TFTs. One is described in [63] and uses a more traditional style of temporal logic (a variation of linear temporal logic (LTL)). The other



(a) Basic event **(b)** Intermediary event **(c)** AND gate



(d) OR gate **(e)** NOT gate **(f)** Transfer symbol

Figure 8 – SFT gates

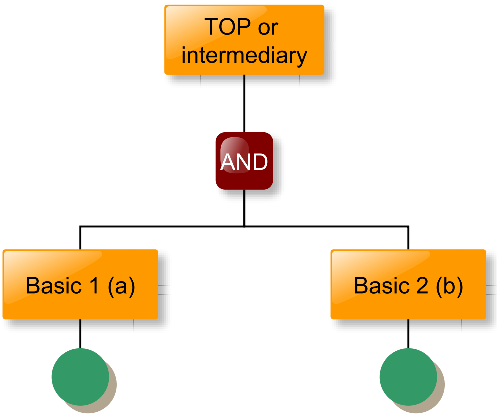


Figure 9 – Very simple example of a fault tree

version is called Pandora and is the one we refer to in the following.

TFTs express ordering of events by directly focusing on ordering relationships rather than different states of a system. Basically they extend SFT's PAND gates, allowing analysis of FT with such gates. It is simpler to express than DFT, but lacks the fault-tolerance-related gate of DFTs (which we show in Subsection 3.1.3).

Structure expressions are also present in TFTs [20, 21, 33], through the Pandora methodology. These expressions use the SFT operators OR and AND, and three new operators[2] related to events ordering: priority-AND (PAND), priority-OR (POR), and simultaneous-AND (SAND). The semantics of the PAND in TFTs is similar to the semantics of the PAND described in the Fault Tree Handbook. To avoid ambiguous expressions, the semantics in TFTs is stated in terms of natural numbers, using a *sequence value* function. For every possible combination of events ordering, it assigns a sequence

---

[2] In formulas, the following symbols are used to represent the operators, respectively: "<", "|", and "&"

Table 1 – TTT of TFT's operators and sequence value numbers

A	B	AND	OR	PAND	POR	SAND
0	0	0	0	0	0	0
0	1	0	1	0	0	0
1	0	0	1	0	1	0
1	1	1	1	0	0	1
1	2	2	1	2	1	0
2	1	2	1	0	0	0

value to each fault event. For example, if event A occurs before event B, then the sequence value of A is lower than the sequence value of B, and one formula to express this is $A < B$.

An invariant on sequence values is that there are no gaps for assigned values. For example, if faults A and B occur at the same time and there are only these two events, then they should both be assigned value 1. On the other hand, if A occurs before B, then the assigned values are 1 and 2, respectively. The possible values increases with the number of variables to express the cases that all events occur in different times, for example, A occurs before B, and B occurs before C. In this case, the assigned values are 1, 2, and 3, respectively. Value zero means that the event is not active on the combination. Similar to Boolean's truth tables, the Pandora methodology defines TTTs. They represent formula values for every combination of events. Table 1 shows the TTT of all TFT operators according to the semantics described in terms of a sequence value function S as follows:

$$S(A \wedge B) = \begin{cases} \max(S(A), S(B)) & \text{if } S(A) > 0 \wedge S(B) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.1a}$$

$$S(A \vee B) = \begin{cases} \min(S(A), S(B)) & \text{if } S(A) > 0 \wedge S(B) > 0 \\ \max(S(A), S(B)), & \text{otherwise} \end{cases} \tag{3.1b}$$

$$S(A < B) = \begin{cases} S(B) & \text{if } S(A) > 0 \wedge S(B) > 0 \wedge S(A) < S(B) \\ 0 & \text{otherwise} \end{cases} \tag{3.1c}$$

$$S(A \mid B) = \begin{cases} S(A) & \text{if } S(A) < S(B) \vee S(B) = 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.1d}$$

$$S(A \,\&\, B) = \begin{cases} S(A) & \text{if } S(A) > 0 \wedge S(B) > 0 \wedge S(A) = S(B) \\ 0 & \text{otherwise} \end{cases} \tag{3.1e}$$

Figure 10 shows TFT-specific symbols used in this work. To illustrate TFTs, for the formula $(A < C) \vee (A \wedge B)$, we show: (i) the TFT in Figure 11, and (ii) its corresponding TTT in Table 2 (the column '#' indicates the MCSeq number).

From structure expressions in order-sensitive FTs (TFT and DFT), MCSeqs are obtained. Several approaches represent MCSeq ordering differently. For the best of our

PAND
POR
SAND

(a) PAND gate **(b)** POR gate **(c)** SAND gate

Figure 10 – TFT-specific gates

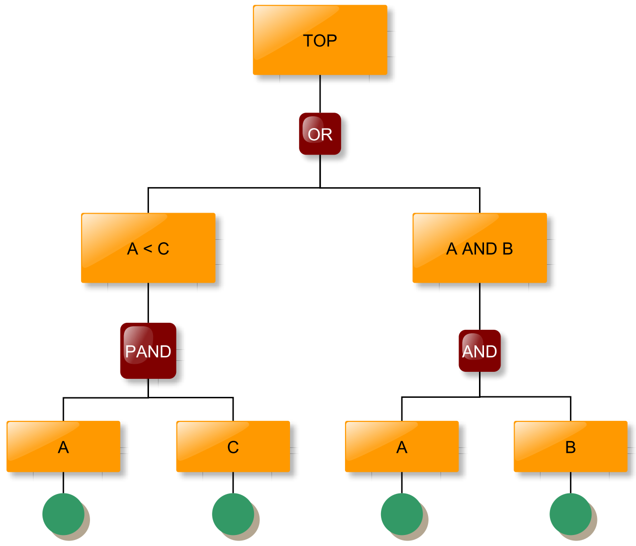


Figure 11 – TFT small example

knowledge they are introduced in the work [64] similarly to MCS, allowing set elements with arrows (“→”) to represent order.

For TFTs, in the work [21] MCSeqs are represented as a DNF using AND and the temporal operators (PAND, POR, and SAND) as doublets (a single temporal relation)—which are the minimal terms—or prime implicants—in the DNF. In a doublet, the expression is a product (AND) of temporal operators, and each temporal operator contains *exactly* two events. The conversion to doublets uses the temporal laws as shown in [21]. For example, the expression $(X \,\&\, Y) \mid Z$ is a temporal relation (POR) of a temporal relation (SAND). To extract MCSeqs it needs to be converted to $[X \,\&\, Y] \wedge [X \mid Z] \wedge [Y \mid Z]$ (the square brackets is the doublets notation and the conversion is the definition of the *Temporal Distributive Law* [21, p. 120]).

The normal form for TFT is similar to that for SFT: it is a DNF with temporal operators (PAND, POR, SAND) in the minimal terms. The reduction of TFT structure expressions is achieved using DT. In a DT, if all children of a tree node are true, then the node is also true. Conversely, if a node is true, then all its children are also true. An issue with DTs is that they grow exponentially. According to the work reported in [33], it is already infeasible to deal with seven fault events in TFTs. Although there is a solution,

Table 2 – TTT of a simple example

#	A	B	C	$A < C$	$A \wedge B$	$(A < C) \vee (A \wedge B)$
01	0	0	0	0	0	**0**
02	0	0	1	0	0	**0**
03	0	1	0	0	0	**0**
04	0	1	1	0	0	**0**
05	0	1	2	0	0	**0**
06	0	2	1	0	0	**0**
07	1	0	0	0	0	**0**
08	1	0	1	0	0	**0**
09	1	0	2	2	0	**2**
10	1	1	0	0	1	**1**
11	1	1	1	0	1	**1**
12	1	1	2	2	1	**1**
13	1	2	1	0	2	**2**
14	1	2	2	2	2	**2**
15	1	2	3	3	2	**2**
16	1	3	2	2	3	**2**
17	2	0	1	0	0	**0**
18	2	1	0	0	2	**2**
19	2	1	1	0	2	**2**
20	2	1	2	0	2	**2**
21	2	1	3	3	2	**2**
22	2	2	1	0	2	**2**
23	2	3	1	0	3	**3**
24	3	1	2	0	3	**3**
25	3	2	1	0	3	**3**

it is based on a mixed application of DTs, modularisation of independent subtrees, and algebraic laws [20]. We show DTs in Subsection 3.2.3. Some of these algebraic laws are:

$$(X < Y) \vee (X \mathbin{\&} Y) \vee (Y < X) = X \wedge Y \qquad \text{Conjunctive Completion Law} \tag{3.2a}$$

$$(X \mid Y) \vee (X \mathbin{\&} Y) \vee (Y \mid X) = X \vee Y \qquad \text{Disjunctive Completion Law} \tag{3.2b}$$

$$(X \mid Y) \vee (X \mathbin{\&} Y) \vee (Y < X) = X \qquad \text{Reductive Completion Law 1st} \tag{3.2c}$$

$$(X \wedge Y) \vee (X \mid Y) = X \qquad \text{Reductive Completion Law 2nd} \tag{3.2d}$$

3.1.3 Dynamic Fault Trees

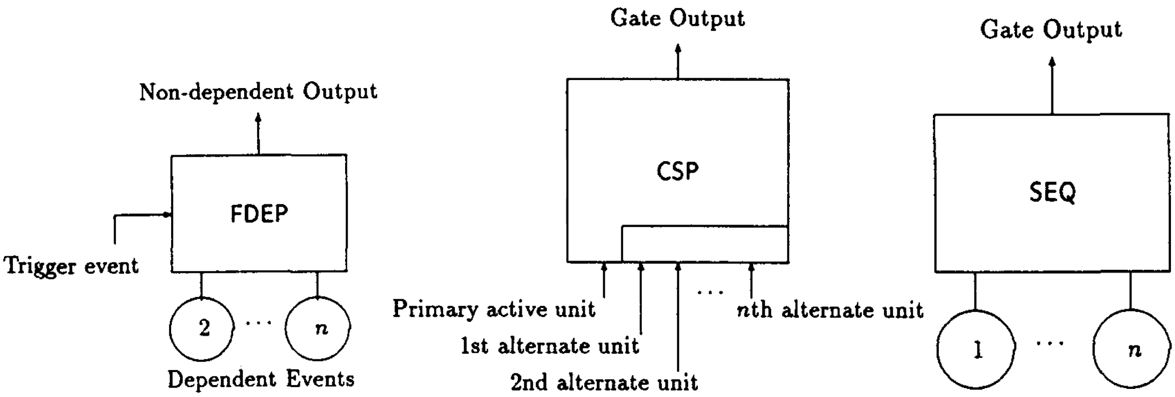
Dynamic Fault Trees were designed with the goal of analysing complex systems with dynamic redundancy management and complex fault and recovery mechanisms [1]. The idea was to create easy-to-use and less error-prone modelling tools than using DTMCs—or simply *Markov chains*—directly. So, since the very beginning, DFTs were intended to be evaluated using Markov chains. Figure 12 depicts the original gate symbols as shown in [1, 2]. In this work, we use gate symbols as depicted in Figure 13. The informal semantics of them are:

FDEP: When the trigger event occurs, the dependent events are forced to occur. Timing in this gate between trigger event and dependent events occurrences can be instantaneous (like in TFT's SAND gate), or a small amount of time, thus implying an order of occurrence, depending on the kind of dependency.

CSp: It is a specific gate to handle spare components. It is important to note that connected inputs are not components—they are fault events of connected components. If the i*th* input is already active (fault has occurred), then it is expected that the input $(i+1)th$ is not, following the specified order. The output becomes true after all connected inputs become true. A spare event can be connected to more than one CSp gate, representing the spare unit connection to one or more components.

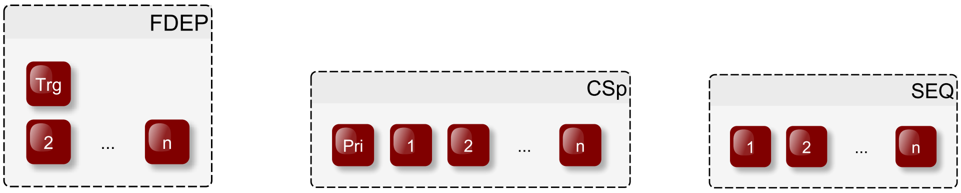
PAND: The same as in TFT: when the connected input events occur in the specified order, it outputs true.

SEQ: The connected events *shall* occur in the specified order. It is different from the PAND gate, because the latter *detects* the specified order. The usage of this gate is usually associated with FDEP.


Non-dependent Output
FDEP
Trigger event
2
n
Dependent Events
Gate Output
CSP
Primary active unit
1st alternate unit
2nd alternate unit
nth alternate unit
Gate Output
SEQ
1
n

(a) FDEP gate (b) CSp gate (c) SEQ gate

Figure 12 – DFTs's original gates symbols


FDEP
Trg
2
...
n
CSp
Pri
1
2
...
n
SEQ
1
2
...
n

(a) FDEP gate (b) CSp gate (c) SEQ gate

Figure 13 – DFTs's gates symbols

There are several means to analyse DFTs qualitative and quantitatively. The works reported in [3, 65, 22, 23] use structure expressions to perform both qualitative and quantitative analysis, and the work reported in [23] summarizes other approaches. We increment their summary (Table 3) and categorize them as:

a) Finding MCSeqs (qualitative analysis) is obtained by replacing DFT gates with SFT gates, using the text as its logical constraints. MCSs in the SFT are

Table 3 – DFT conversion to calculate probability of top-level event

Conversion	Calculation	Explained in
Automaton-like structure	CTMC	[35]
Bayesian network (BN) [66]	Inference algorithm (model-specific)	[36]
Stochastic well-formed net (SWN) [67] (a kind of coloured Petri-net (CPN) [68])	CTMC	[69]
SBDD (a modified version of BDD)	model-specific	[42, 43]

expanded using timing constraints from the texts into MCSeq. In this case, the behaviour of spare events cannot be correctly taken into account;

b) Quantitative analysis consists in converting a DFT to a well-defined formalism to calculate the probability of its top-level event. Table 3 shows the conversion options, the calculation, and where the method is explained.

In [3, 65, 22] fault events occur in a specific time and are instantaneous (similar to detected faults), stated through a "date-of-occurrence" function. As the "date-of-occurrence" function is stated in continuous time, the probability of two events occurring at the same time is negligible. In fact, useful information is obtained from the possibilities of relation in time of the occurrence of the events. DFT gates' algebraic model is summarized in Table 4. Structure expressions are written with an algebra that has operators OR and AND, and three new operators to express events ordering: (i) non-inclusive-before (NIBefore), (ii) simultaneous (SIMLT), and (iii) inclusive-before (IBefore). The NIBefore and the SIMLT operators are similar to TFT's POR and SAND operators, respectively. The IBefore is a composition of NIBefore and SIMLT operators. Table 5 summarizes the date-of-occurrence function for all operators. An infinite value means the event never occurs.

MCSeqs are extracted from canonical form of structure expressions written in a DNF. Minimal terms are products of variables and NIBefore operators (the other operators can be written as combinations of NIBefore). The reduction of DFT structure expressions uses algebraic laws as, for example:

$$(a \lhd b) \vee (a \bigtriangleup b) \vee (b \lhd a) = a \vee b \tag{3.3a}$$

$$(a \wedge (b \lhd a)) \vee (a \bigtriangleup b) \vee (b \wedge (a \lhd b)) = a \wedge b \tag{3.3b}$$

$$(a \unlhd b) \wedge (b \unlhd a) = a \bigtriangleup b \tag{3.3c}$$

Figure 14 shows an example of DFT extracted from [23]. It is a cardiac assist system (HCAS), which is divided in four modules: trigger, CPU unit, motor section, and pumps. The trigger is divided in two components, CS and SS. The failure of any CS or SS,

Table 4 – Algebraic model of DFT gates with inputs A and B

Gate	**Algebraic model of gate's output**	**Note**
FDEP	$A_T = T \vee A$ and $B_T = T \vee B$	A_T and B_T replace A and B on the resulting expression
CSp	$(B_a \wedge (A \lhd B_a)) \vee (A \wedge (B_d \lhd A))$	A is the active input, and B is the spare. Subscripts a and d represent component's state—*active* and *dormant*, respectively, which are used on the failure distribution formulas
PAND	$B \wedge (A \unlhd B)$	No distinction of active or dormant states.

Table 5 – Date-of-occurrence function for operators defined in [3]

Operator	**Expression**	**Expr. value if** $\mathbf{d}(\mathbf{a}) < \mathbf{d}(\mathbf{b})$	**Expr. value if** $\mathbf{d}(\mathbf{a}) = \mathbf{d}(\mathbf{b})$	**Expr. value if** $\mathbf{d}(\mathbf{a}) > \mathbf{d}(\mathbf{b})$
OR	$d(a \vee b)$	$d(a)$	$d(a)$	$d(b)$
AND	$d(a \wedge b)$	$d(b)$	$d(a)$	$d(a)$
NIBefore	$d(a \lhd b)$	$d(a)$	$+\infty$	$+\infty$
SIMLT	$d(a \vartriangle b)$	$+\infty$	$d(a)$	$+\infty$
IBefore	$d(a \unlhd b)$	$d(a)$	$d(a)$	$+\infty$

triggers a CPU unit failure. The primary CPU (P) has a warm[3] spare (B). The motor module fails if both M and MC fail. In order for the pumps unit to fail, all three pumps need to fail, and the left-hand side spare gate needs to fail before (or at the same time as) the right-hand side spare gate (PAND gate[4]). The top-level event structure expression is:

$$\begin{aligned} SYSTEM = {} & CS \vee SS \vee (M \wedge MC) \vee \\ & (P \wedge (B_d \lhd P)) \vee (B_a \wedge (P \lhd B_a)) \vee \\ & (BP_a \wedge (P2 \lhd P1) \wedge (P1 \lhd BP_a)) \vee (P2 \wedge (P1 \lhd BP_a) \wedge (BP_a \lhd P2)) \end{aligned} \tag{3.4}$$

3.2 Structure expression analysis

In this section we explain the difference of stateful and stateless methods to analyse structure expressions and detail the stateless methods. Another common approach to analyse an FT is to perform structure expression analysis based on algebraic laws. Boolean laws are well-known and are used for SFTs, temporal laws [21, 33] for TFTs, and the works reported in [3, 22] show laws for DFTs. An issue with algebraic laws is that, in some cases, the expression needs to be expanded before it gets reduced, so reduction automation is not

---

[3] Warm spare gates only differ from CSp on the activation time.

[4] Although the original example uses a PAND gate, accordingly to the informal description, a SEQ gate would fit better.

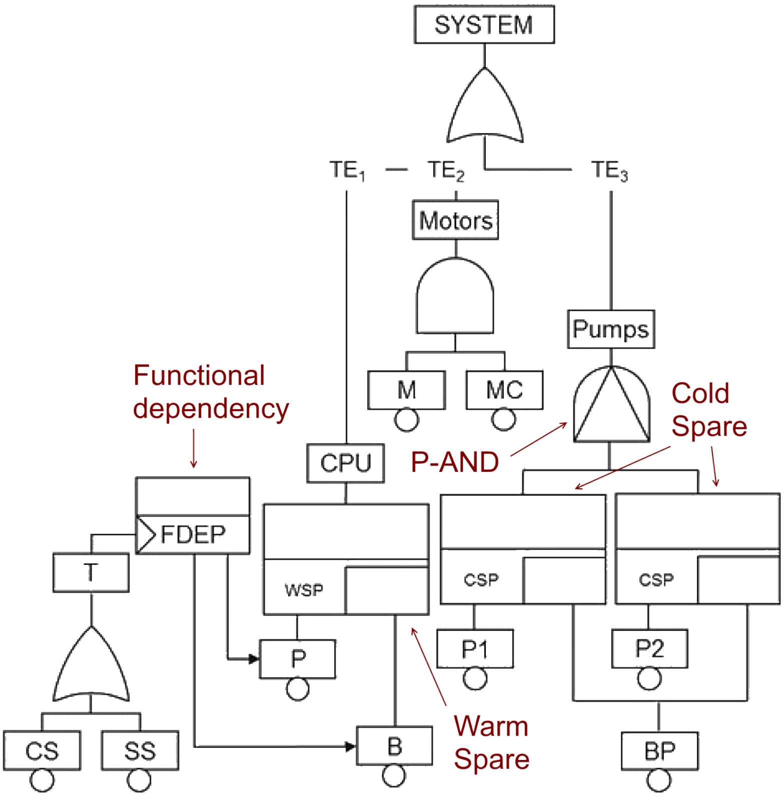


Figure 14 – DFT example

trivial without a theorem prover. For example, the following TFT's structure expression needs to be expanded [33] before it gets reduced:

$$\begin{aligned}
&(X \wedge Y) \vee ((X < Y) \wedge Z) \\
= &(X < Y) \vee (X \,\&\, Y) \vee (Y < X) \vee ((X < Y) \wedge Z) && \text{by Eq. (3.2a)} \\
= &(X < Y) \vee (X \,\&\, Y) \vee (Y < X) && \text{by Boolean absorption of } X < Y \\
= &X \wedge Y && \text{by Eq. (3.2a)}
\end{aligned}$$

A denotational semantics to Boolean expressions—and consequently to SFT—is given by FBAs (Section 3.3). Using denotational semantics in favour of axiomatic laws is preferred as the axioms must be very simple or must be generally accepted, because a subtlety in an axiom (an unfounded axiom) may invalidate a whole theory.

There are several works with stateful analysis methods for FTs (SFT, TFT, and DFT). We show some of them in Subsection 3.2.1.

3.2.1 Stateful methods and temporal logic analysis

The work reported in [70] shows a formal approach to analyse SFT using Interval Temporal Logic (ITL) [71]. Instead of tackling basic events ordering (as in PAND), it considers a causal relation over a gate, as for example, a relation of a basic event and a higher-level intermediary event.

For TFTs, the works reported in [72, 73] show an inverse solution. They map Finite State Machines (FSMs) to Pandora logic, then verify system properties. They show that such a mapping simplifies expression reduction, thus improving performance on the analysis.

Although there is formal modelling approaches to DFTs, they do not implement a direct modelling of the DFT itself. Instead, most of the works propose a formal modelling using a state-based approach. The work reported in [35] shows a formal model of Markov chains in the Z Notation (Z) [74] and each DFT element (basic events and gates). The analysis uses a quantifier on states of the resulting Markov chain automaton. The work reported in [75] shows a methodology to perform a modular analysis of DFTs based on BDD and Markov chain. As DFT extends SFT, it identifies subtrees that are purely SFT and uses BDD, otherwise. It performs Markov chain analysis. Still on the state-based approaches, the work reported in [76] maps DFTs to high-level Petri-net (HLPN) [77] to analyse false alarms.

In the following we show specific stateless methods that are designed to reduce structure expressions. In essence, the methods are very similar. Structure expressions for SFTs can be reduced using BDDs (Subsection 3.2.2), TFTs can be reduced using DTs (Subsection 3.2.3), MCSeqs of DFTs can be obtained using Zero-suppresed Binary Decision Diagram (ZBDD) [64] (Subsection 3.2.4), and the works reported in [42, 43] show the analysis of standby systems (CSp gates) using SBDDs (Subsection 3.2.5).

3.2.2 Binary Decision Diagrams

BDDs are directed acyclic graphs that represent a Boolean expression. They are still referred to as BDD, but the more spread version is the Reduced Ordered Binary Decision Diagram (ROBDD) [78], which is an optimisation. There are two ways to generate a BDD for an expression: (i) derive a diagram from the truth-table, or (ii) expand the paths based on Shannon's method (described in the Fault Tree Handbook).

To demonstrate the expressiveness of a BDD, Figure 15 shows a diagram for a truth table with three variables (Table 6). In a node, when a path is chosen, the variable of the node assumes the edge value. For example, the top-level node variable of Figure 15 is A. Following the right-hand side of the node, all leaf nodes correspond to the lines of the truth table that A has "0" values (the first four lines). The symbol nodes are replaced by the values assumed by a specific formula.

Following Shannon's method, we choose a variable and build the lower level BDD assuming the edge value for the chosen variable. In the remainder of the path, the variable's value is unchanged. For example, the expression $A \vee (\neg B \wedge C)$ has value "0" in the lines a and c, and value "1" in the other lines. By choosing the variable A first, then B and

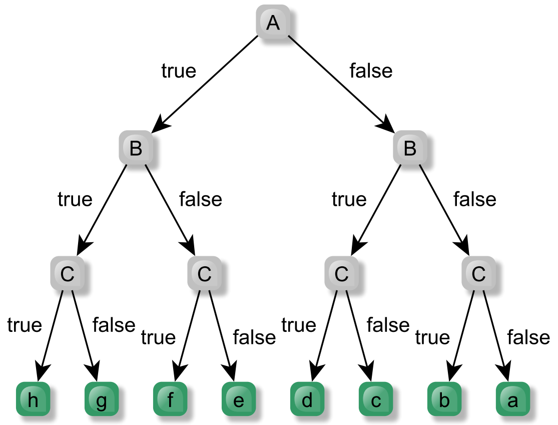

A
true
false
B
B
true
false
true
false
C
C
C
C
true
false
true
false
true
false
true
false
h
g
f
e
d
c
b
a

Figure 15 – A diagram for a truth table

Table 6 – Truth table for a formula outputs with three variables (A, B, and C)

A	B	C	Formula
0	0	0	a
0	0	1	b
0	1	0	c
0	1	1	d
1	0	0	e
1	0	1	f
1	1	0	g
1	1	1	h

C, the resulting BDD with the binary values nodes (called sink nodes "false" and "true") for this formula is depicted in Figure 16. Starting from the top-level node A, the formula expressed by the BDD is true if A assumes value true. If A is false, and B is false, the expression is only true if C is true.

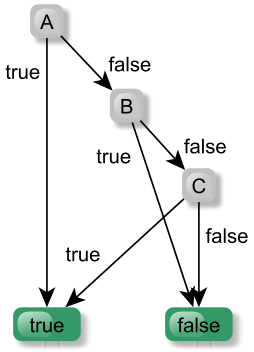

A
true
false
B
true
false
C
false
true
true
false

Figure 16 – A BDD for the expression $A \lor (\neg B \land C)$

Figure 16 is an ROBDD. To be considered an ROBDD, the BDD must meet the following constraints [78]:

a) the variables are assigned a constant ordering;

b) every path to sink nodes visit the input variables in ascending order;

c) each node represents a distinct logic function.

The size of an ROBDD, for a given expression, depends on the chosen variables ordering. The work reported in [79] shows initial findings on variable ordering, and the work reported in [80] shows heuristics to improve the performance for optimal order search.

For SFTs the evaluation of a BDD is the calculation of the probability of the paths ending in *true*. For example, the probability of the expression in Figure 16 is obtained from the formula: $\Pr\{A \vee (\neg A \wedge \neg B \wedge C)\}$. Note that the formula in the probability calculation is different from the formula that originated the diagram.

3.2.3 Dependency tree

Dependency tree (DT) [33] is a hierarchical acyclic graph of expressions that shows all possible cut sequences for any given set of events. It is a graphical view of a TTT. At the top of a DT are the variables, that is, the single events that occur in an expression. On the lower levels are the increasingly complex expressions. Each node represents an MCSeq. Figure 17 shows a DT with all nodes for variables X and Y.

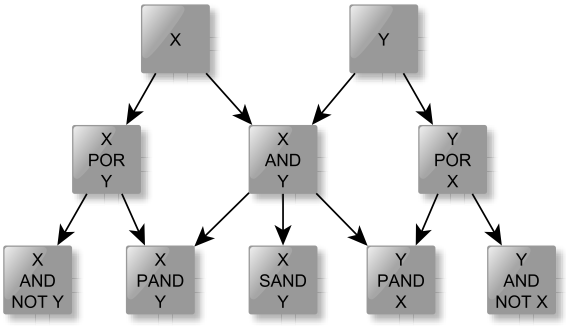


Figure 17 – DT for variables X and Y

The reduction of a structure expression is given by the activation (true values) of nodes. If a node is active (true), then all child nodes are also active; the converse is also true: if all node's children are active, then it is also active. The reduced expression is given by the DNF created with the expressions of higher active level nodes. To reduce the formula $(X \wedge Y) \vee ((X < Y) \wedge Z)$, given on the beginning of this subsection, we create the DT depicted in Figure 18. Nodes marked with "1" are those MCSeqs given directly by the formula. Nodes marked with "2" are child nodes of the "1"'s nodes, and so forth. The node of the expression $((X < Y) \wedge Z)$ is a grandchild of $X \wedge Y$ and thus it is not necessary. The final expression is obtained by the active higher level node, which is $X \wedge Y$.

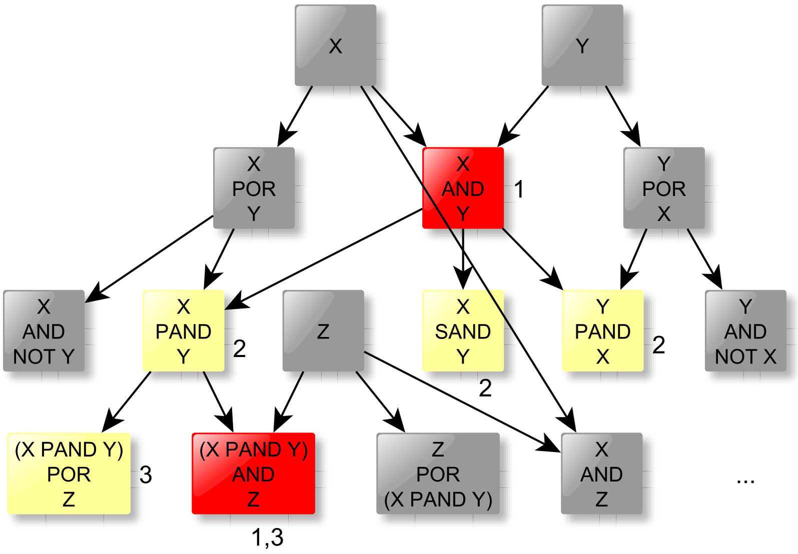


Figure 18 – DT for the formula $(X \wedge Y) \vee ((X < Y) \wedge Z)$

3.2.4 Zero-suppresed Binary Decision Diagrams

The work reported in [64] proposes Zero-suppresed Binary Decision Diagram, which is a variant of BDD, and uses set manipulations (union, intersection, difference, and product) to obtain MCSeqs of DFTs.

To reduce a BDD to a ZBDD, the nodes that have the "true" ('1') path pointing to the "false" ('0') sink node are removed, and the parent node is connected directly to the "false" subgraph of the removed node. Figure 19 shows an example of ZBDD for the combination set $\{a, b\}$, as shown in [64]. The idea of the reduction is to remove irrelevant variables and nodes. The irrelevant variables are set to "false". The method obtains the MCSeqs by navigating the paths to sink node "true".

Although the work reported in [64] shows ZBDD, the final solution does not use them directly. Instead, it defines a hierarchical manipulation of DFT to obtain the MCSeqs when traversing the a DFT. The order-related operators in a DFT are replaced by a new event, which takes ordering into account. The idea is to transform the DFT into an SFT, in a very similar way as the one shown in [42]. Finally, the MCSeqs are obtained using set manipulation with elements that are basic events alone or order-related operators. These order-related operators are event-to-event only, so they cannot be combined with other sets.

The use of sets in [64] is very related to our ATF. We use sets of sequences to define the ATF, but keep the analysis with set operators. In ATF we do not create new events that represent an order-related operator. Our order-related operator has a set-based semantics that can be combined with other non-order-related (Boolean) operators.

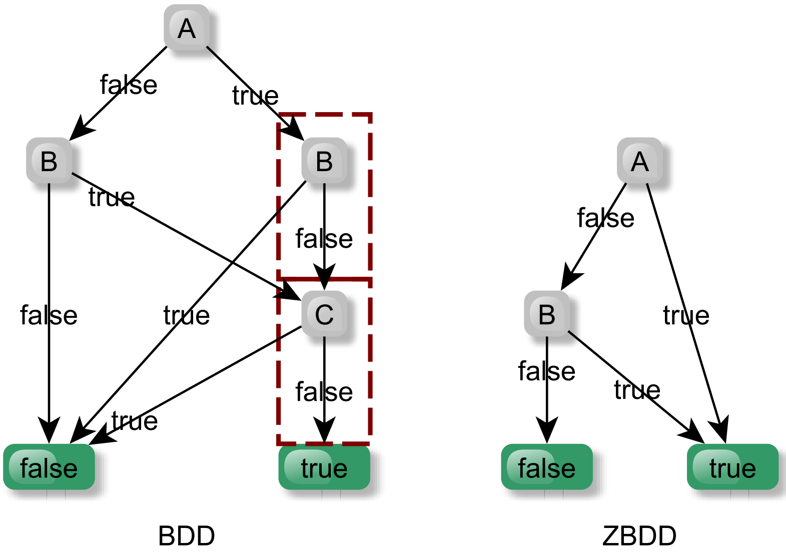


Figure 19 – ZBDD example of combination set $\{a, b\}$

3.2.5 Sequential Binary Decision Diagrams

SBDD is an extension of BDD to tackle ordering of events in DFTs for CSp and WSp gates. Ordering of events in CSp gates [43] is slightly different from WSp [42]. A backup system in CSp gets activated slower than in WSp, which implies that there are less failure possibilities in CSp, but its the readiness is lower than in WSp. SBDD adds a new node kind that contains a binary operation of fault events, which allows to express the ordering of events. One kind of operation expresses the slowness of the relation of the fault events of CSp, and another one expresses the readiness of the WSp. The latter semantics is similar to the semantics of PAND and IBefore (combined with AND) gates.

SBDD creation has two steps: (i) CSp or WSp DFT conversion, and (ii) SBDD model generation. In (i), it is a DFT-to-DFT conversion. CSp and WSp gates are converted to a new, but equivalent DFT without CSp and WSp gates, where the operations appear as basic events and are combined using other gates. In (ii), the SBDD model is created. The model may contain nodes that are contradictory as, for example, nodes that assumes that an event A is false and a binary operation that contains A is true. This step ends when all contradictions are removed. The evaluation is similar to BDD's: each path ending in true is a minimal term in the DNF that may contain one of the binary operations and individual events.

3.3 Free Boolean Algebras

Another means to analyse SFTs is to use an FBA to perform set-theoretical operations (intersection, difference, etc.) to reduce expressions. In this section we briefly present the FBA theory and its elements.

Instead of using an axiomatic definition of a Boolean algebra directly, we follow its set-theoretical definition, as shown in [81, pp. 254–258] and [16, pp. 8–11]. This definition represents a Boolean algebra as an algebra of sets and does not rely on Boolean axioms (which can be misleading, case there is an unfounded axiom).

Definition 3.1 (Boolean Algebra)**.** *A Boolean algebra is defined as a triple* $\langle B, \cap, - \rangle$*, where* B *is a set with at least two elements,* $\cap$ *is the intersection (also called meet or* infimum*) and* $-$ *is the complement (also called negation).*

The other Boolean elements (union, $\bot$, and $\top$) are derived from the previous two operators:

$\cup$ is the union (also called join or *supremum*): $A \cup B = -(-A \cap -B)$

$\bot$ is the bottom (also called zero): $\bot = A \cap -A$

$\top$ is the top (also called unit): $\top = -\bot$

A Free Boolean Algebra is defined from a set E of generators. A generator can be represented as a proposition in statement calculus [81, p. 274]. For example, "valve A is stuck closed" and "motor M is malfunctioning" are valid statements. A Free Boolean Algebra is constructed from $\mathbb{P}(E)$, where $\mathbb{P}$ is the power set. Note that if E has n symbols, $\mathbb{P}(E)$ has 2^n elements, called *atoms* of a finite Boolean algebra. For the two statements above, the atoms are:

a) "Valve A is stuck closed" and "motor X is malfunctioning"

b) "Valve A is stuck closed" and "motor X is *not* malfunctioning"

c) "Valve A is *not* stuck closed" and "motor X is malfunctioning"

d) "Valve A is *not* stuck closed" and "motor X is *not* malfunctioning"

Such a Boolean algebra has 2^{2^n} formulas [16, p. 261]. For example, if $E = \{a, b\}$, then $\mathbb{P}(E) = \{\{\}, \{a\}, \{b\}, \{a, b\}\}$, hence the Boolean algebra generated by E contains sixteen (2^{2^2}) formulas: $\{\}$, $\{\{\}\}$, $\{\{\}, \{a\}\}$, $\{\{\}, \{b\}\}, \ldots$, $\{\{a\}, \{a, b\}\}, \ldots$, $\{\{b\}, \{a, b\}\}, \ldots$, $\{\{\}, \{a\}, \{b\}, \{a, b\}\}$.

The Boolean algebra B can be inductively defined using some constructs.

Definition 3.2 (Inductive Free Boolean Algebra)**.** *Let* s *be a statement, then:*

$$\mathbf{var}\, s = \{X | s \in X\} \implies \mathbf{var}\, s \in B \qquad \textit{(variable)} \qquad (3.5a)$$

$$X \in B \implies -X \in B \qquad \textit{(complement)} \qquad (3.5b)$$

$$X \in B \wedge Y \in B \implies X \cap Y \in B \qquad \textit{(intersection)} \qquad (3.5c)$$

The characterisation of a “free” Boolean algebra comes from that, for some valuation function a, some of the formulas evaluates to “1”. Given a function $p : B \times \{0,1\} \to B$, such that:

$$p\,(i,j) = \begin{cases} i & j = 1 \\ -i & j = 0 \end{cases} \tag{3.6}$$

Lemma 3.1 (Free generators (valuation))**.** *Let F be a finite set, such that $F \subseteq E$, and $a : F \to \{0,1\}$, a necessary and sufficient condition for a set E of generators of a Boolean algebra B to be free is then:*

$$\bigwedge_{i \in F} p\,(i, a\,(i)) \neq 0 \tag{3.7}$$

Essentially, Lemma 3.1 states that there is no relation between generators, such as $a = -b$.

Lemma 3.2 (Free generators (algebraic))**.** *Let i and j be statements, such that $i, j \in E$, hence from Definition 3.2 and Lemma 3.1 it is necessary and sufficient that:*

$$\mathbf{var}\,i = \mathbf{var}\,j \iff i = j \tag{3.8a}$$

$$\mathbf{var}\,i \neq -\mathbf{var}\,j \tag{3.8b}$$

$$-\mathbf{var}\,i \neq \mathbf{var}\,j \tag{3.8c}$$

3.4 Using the NOT operator in Static Fault Trees

Although the Fault Tree Handbook introduces several gates, the vast majority of SFT analyses would fit in FTs with only AND and OR gates (coherent FTs). Qualitative analysis requires the reduction of the structure expression of FTs and, when NOT gates are present (non-coherent FTs), such a reduction can cause the interpretation of failure expression to be misled [9, 11, 10, 12, 13]. The work reported in [11] shows three funny examples of this kind of problem, and the works reported in [9, 11, 12] show how to solve it using BDDs. In the following we show: (i) the second example presented in [11], which highlights the problem when using NOT gates (Subsection 3.4.1), and (ii) the second example presented in [9], which defends the usefulness of NOT gates in a multitasking system (Subsection 3.4.2).

Negated events in a non-coherent analysis are in fact the working state of a component. The failure probability contribution of a negated basic event is close to 1. The problem with non-coherent FTs is that its analysis can cause impossible situations. The general formula to identify coherency is given in [9, 12] in terms of a structure function.

Definition 3.3 (FT Coherency)**.** *Let $\Phi\,(x) : B^n \to B^1$ be a binary function of a vector of binary variables, such that $x = [x_1, x_2, \ldots, x_n]$, representing the states of n system's components.*

A binary structure function $\Phi(x)$ is coherent if all the following hold:

a) $\Phi(x)$ *is monotonic (non-decreasing) in each variable;*

b) Each x_i *is relevant, which means that* $\Phi(x)[x_i/1] \neq \Phi(x)[x_i/0]$ *for some vector* x*.*

where $B^1 = \{0,1\}$, $B^n = B^{n-1} \times B^1$, $x_i = 1$ implies that component i failed, and $\Phi(x) = 1$ implies the system failed. For $y = [y_1, y_2, \ldots, y_n]$, monotonicity of Φ means that for *all* i, $x_i \geq y_i$ ($y_i = 1 \implies x_i = 1$), and for *some* i, $x_i > y_i$ ($x_i = 1$ and $y_i = 0$). Variable replacement ($[a/b]$) is as usual: $x[x_i/a] = [x_1, \ldots, x_{i-1}, a, x_{i+1}, \ldots, x_n]$

3.4.1 Non-coherent fault tree misleads

In this subsection we illustrate—with the second example detailed in [11]—how non-coherent FT misleads.

A college student who wants to visit her mother in another city has two options: wake up early (x_3) and take a ride with a friend (x_1), or wake up late ($\neg x_3$) and take the metro (x_2). The top-event failure is "visit mother" with expression $S = (x_1 \wedge x_3) \vee (x_2 \wedge \neg x_3)$. Its fault tree is depicted in Figure 20. It is clear that the structure function is non-coherent in x_3 accordingly to Definition 3.3: $\Phi(1,1,x_3)[x_3/1] = \Phi(1,1,x_3)[x_3/0]$.

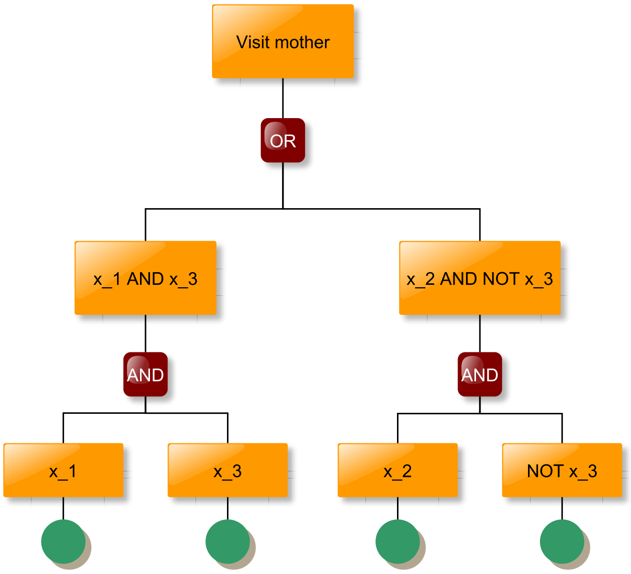


Figure 20 – Non-coherent FT college student's example

The problem with this tree is the interpretation of the qualitative results. One of the possibilities in this scenario is that the college student would take a ride AND take the metro ($x_1 \wedge x_2$). Quantitatively, the analysis of the probabilities shows that this result is not negligible, but its interpretation is impossible.

3.4.2 Usefulness of NOT gates in FTA

In this subsection we show the second example detailed in [9].

The gas detection system depicted in Figure 21 has two sensors D_1 and D_2 which are used to detect a leakage in a confined space. When a leakage is detected, these sensors send a signal to the logic control unit LU, which performs three tasks:

a) shuts-down the main system (process isolation) by de-energizing relay R_1;

b) informs the operator of the leakage by lamp and siren L;

c) deactivates all possible ignition sources, which is the interruption of power supply by de-energizing relay R_2.

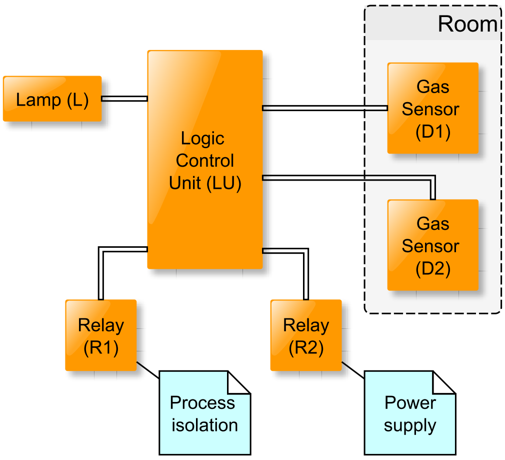


Figure 21 – Gas detection system

The system is in a fail state if it does not perform one of these three tasks. The fault tree that represents this generic failure is depicted in Figure 22. G_1, G_2, and G_3 are subtrees that represents the three tasks "Operator not informed", "Process shut-down fails", and "Power supply not isolated", respectively. All three tasks will fail if their respective main component fails (L, R_1, and R_2) or there is no signal from LU (LU fails or both D_1 and D_2 fail). The structure expressions for the subtrees are:

$$G_1 = L \vee LU \vee (D_1 \wedge D_2)$$
$$G_2 = R_1 \vee LU \vee (D_1 \wedge D_2)$$
$$G_3 = R_2 \vee LU \vee (D_1 \wedge D_2)$$

Analysing in more detail, there are different degrees of system failure. There are eight outcomes (given the three tasks) and the most critical one is when both process shut-down (G_2) and power supply isolation (G_3) fail keeping energized upon a leakage, and the operator is not informed (G_1), but the operator information system is working (lamp and siren are off, but they are operational). The coherent FT of this outcome is depicted

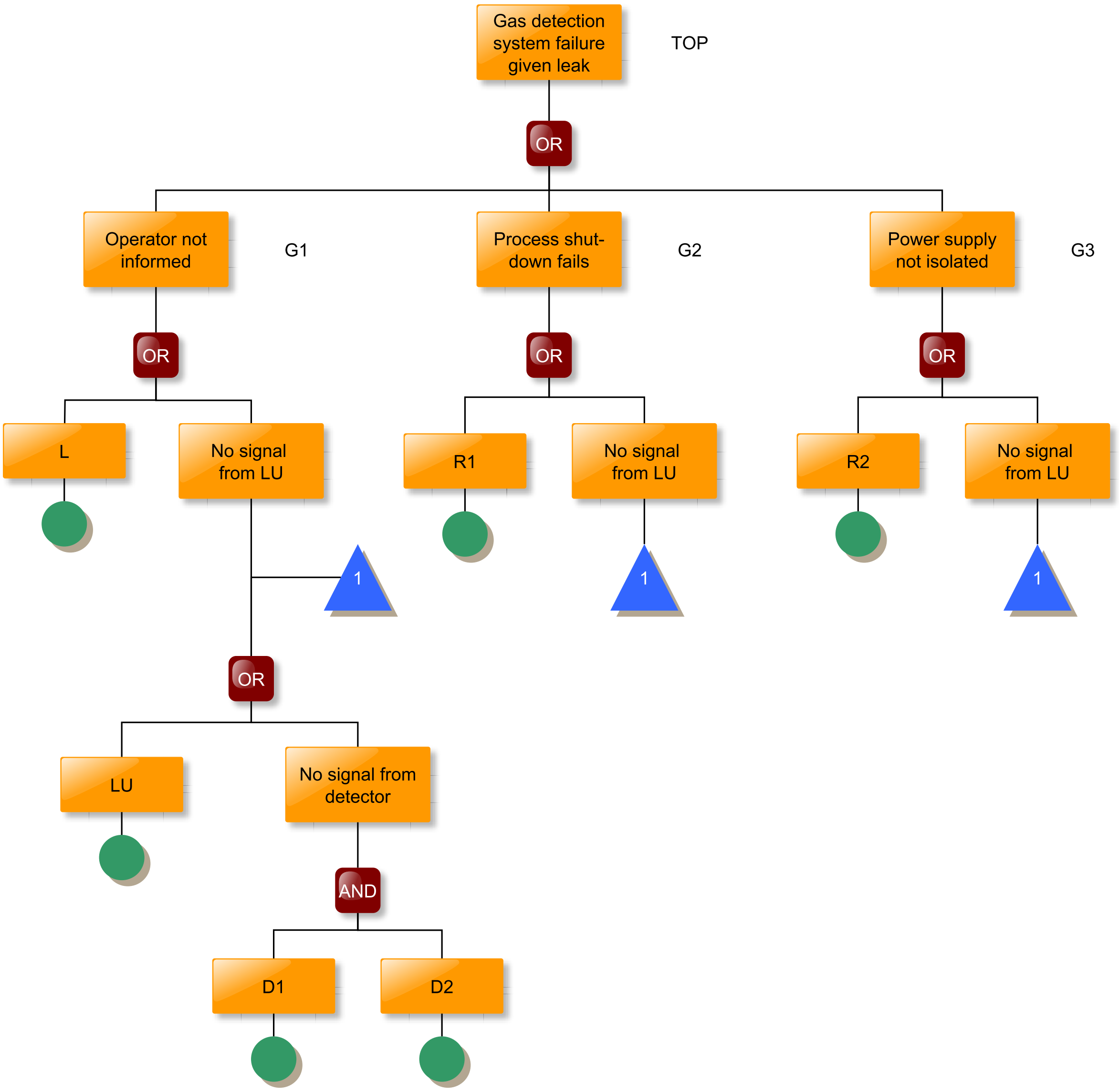

Gas detection system failure given leak
TOP
OR
Operator not informed
G1
Process shut-down fails
G2
Power supply not isolated
G3
OR
OR
OR
L
No signal from LU
R1
No signal from LU
R2
No signal from LU
1
1
1
OR
LU
No signal from detector
AND
D1
D2

Figure 22 – FT for a generic failure in the gas detection system

in Figure 23. The minimal cut sets obtained from this will be: $\{R_1, R_2\}$, $\{D_1, D_2\}$, and $\{LU\}$.

Quantification of the coherent FT will overestimate the probability of the critical outcome unless the part of the system that is working (lamp and siren L, LU, and sensors D_1 and D_2) is taken into account. The non-coherent FT with the working part is shown in Figure 24.

If the operator *can* be informed, then cut sets $\{D_1, D_2\}$ and $\{LU\}$ could not have occurred (see Figure 22), thus the correct qualitative analysis should consider only cut set $\{R_1, R_2\}$. Reducing the expressions of the non-coherent FT (Figure 24), we obtain the structure expression: $\neg L \wedge \neg LU \wedge R_1 \wedge R_2 \wedge (\neg D_1 \vee \neg D_2)$. The approximation for this expression, removing the negated events, gives the cut set $\{R_1, R_2\}$, which gives a correct

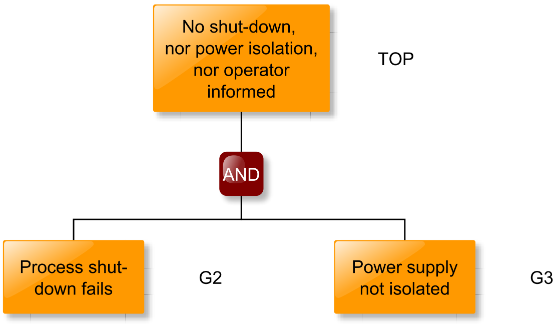

No shut-down, nor power isolation, nor operator informed
TOP
AND
Process shut-down fails
G2
Power supply not isolated
G3

Figure 23 – *Coherent* FT for the most critical outcome of the gas detection system

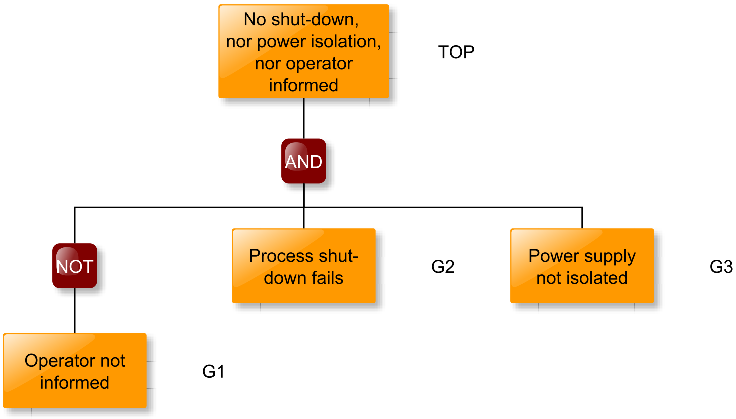

No shut-down, nor power isolation, nor operator informed
TOP
AND
NOT
Process shut-down fails
G2
Power supply not isolated
G3
Operator not informed
G1

Figure 24 – *Non-coherent* FT for the most critical outcome of the gas detection system

quantitative analysis.

3.5 Systems nominal model and fault injection

Control system modelling using Simulink block diagrams [82] is recommended in [30] and have been used by our industrial partner. It is a complementary tool of Matlab [83]. In fact, it works as a graphical interface to Matlab. A Simulink model has blocks and connections between these blocks, named signals. Each block has inputs and outputs and an internal behaviour expressed by its mathematical formula, which defines a function of the inputs for each output. There are many predefined blocks in the tool. It is also possible to create new blocks or use subsystems that encapsulate other blocks. A simulation adds extra parameters to a block diagram, like elapsed time and time between states. The elapsed time of a simulation is an abstraction for the quantity of possible simulation states and the time between states is related to the lowest common denominator of the sample time. Some components define different sample times, depending on their mode of operation. Usually, the value for this property is set to *auto*, allowing Simulink to choose a proper value automatically.

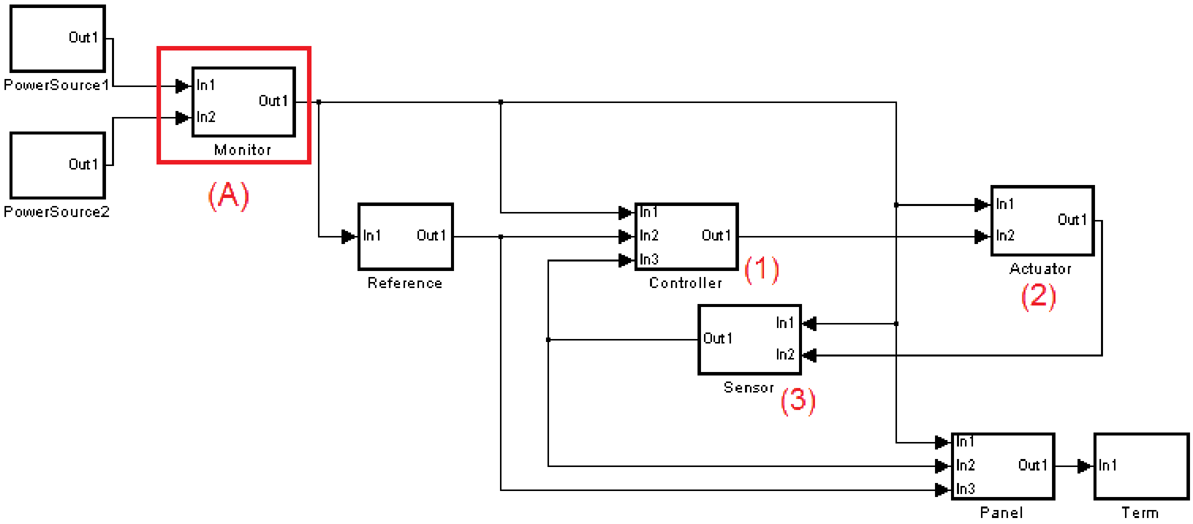

Out1
PowerSource1
Out1
PowerSource2
In1
In2
Out1
Monitor
(A)
In1
Out1
Reference
In1
In2
In3
Out1
Controller
(1)
In1
In2
Out1
Actuator
(2)
Out1
In1
In2
Sensor
(3)
In1
In2
In3
Out1
Panel
In1
Term

Figure 25 – Block diagram of the ACS provided by our industrial partner (nominal model)

Nowadays, control systems are usually composed of an electromechanical part and a processor. Figure 25 shows the components of a feedback system [84] which was provided by our industrial partner. In this system, the feedback behaviour is given by the *Controller* (1), *Actuator* (2) and *Sensor* (3). A command is received by the *Controller*, which sends a signal to the *Actuator* to start its movement. The *Sensor* detects the actual position of the *Actuator* and sends it back to the *Controller*, which adjusts the given command to achieve the desired position. This loop (feedback) continues until the desired position given by the original command is reached.

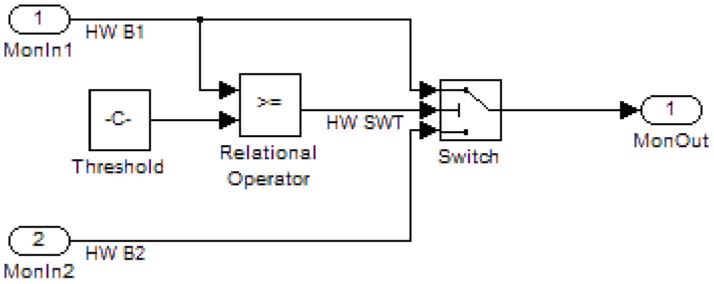

1
MonIn1
HW B1
-C-
Threshold
>=
Relational Operator
HW SWT
Switch
1
MonOut
2
MonIn2
HW B2

Figure 26 – Internal diagram of the monitor component (Figure 25 (A)).

Figure 26 shows the internal elements of the monitor component (Figure 25 (A)), which is used as a case study in Chapter 6 to illustrate our strategy. The outputs of the hardware elements are annotated with HW, which are the two power sources and an internal component of the monitor (switch command).

To perform a formal verification in a Simulink system model we use a model-checking tool, FDR. It is a refinement checker for formal models written in CSP_M. To verify a refinement, it takes two specifications: (i) a specification with more abstract properties, and (ii) an implementation with more concrete properties. If a refinement does not hold (the implementation fails to refine the specification), FDR shows counter-examples as traces of events. The CSP_M language is suitable to model concurrent behaviour and is very expressive to model systems' states. The work reported in [31] translates a Simulink model to the CSP_M language. The resulting CSP_M code (implementation) is then used to check if it meets functional requirements also encoded in CSP_M (specification).

In our previous work, reported in [28], we modified such a translation to perform fault injection using hardware annotations allowing a subsystem or part to "break" randomly. We designed a CSP_M process to act as an observer (specification), watching outputs of the nominal version and comparing to the outputs of the "breakable" version (with injected faults—the implementation) of the system. When the CSP_M process of the model and the observer are loaded into the FDR model-checker, counter-examples are generated for each output that differs from the nominal model, thus obtaining a *sequence* of injected fault combinations that leads to the unexpected output, which are indeed *fault traces*.

In what follows, injected faults and the top-level failure have generic names based on the names of the Simulink model blocks. It is out of the scope of [28] to define event names.

For the Simulink model shown in Figure 26, some representative fault traces are:

```
TRACE 1:
failure.Hardware.N04_RelationalOperator.1.EXP.B.true
failure.Hardware.N04_RelationalOperator.1.ACT.B.false
failure.Hardware.N04_MonIn2.1.EXP.I.5
failure.Hardware.N04_MonIn2.1.ACT.OMISSION
out.1.OMISSION

TRACE 2:
failure.Hardware.N04_MonIn2.1.EXP.I.5
failure.Hardware.N04_MonIn2.1.ACT.OMISSION
failure.Hardware.N04_RelationalOperator.1.EXP.B.true
failure.Hardware.N04_RelationalOperator.1.ACT.B.false
out.1.OMISSION

TRACE 3:
failure.Hardware.N04_MonIn1.1.EXP.I.5
failure.Hardware.N04_MonIn1.1.ACT.OMISSION
failure.Hardware.N04_MonIn2.1.EXP.I.5
failure.Hardware.N04_MonIn2.1.ACT.OMISSION
out.1.OMISSION

TRACE 4:
failure.Hardware.N04_MonIn2.1.EXP.I.5
failure.Hardware.N04_MonIn2.1.ACT.OMISSION
failure.Hardware.N04_MonIn1.1.EXP.I.5
failure.Hardware.N04_MonIn1.1.ACT.OMISSION
out.1.OMISSION
```

```
TRACE 5:
failure.Hardware.N04_MonIn1.1.EXP.I.5
failure.Hardware.N04_MonIn1.1.ACT.OMISSION
failure.Hardware.N04_RelationalOperator.1.EXP.B.false
failure.Hardware.N04_RelationalOperator.1.ACT.B.true
out.1.OMISSION

TRACE 6:
failure.Hardware.N04_MonIn1.1.EXP.I.5
failure.Hardware.N04_MonIn1.1.ACT.OMISSION
failure.Hardware.N04_RelationalOperator.1.EXP.B.false
failure.Hardware.N04_RelationalOperator.1.ACT.B.true
failure.Hardware.N04_MonIn2.1.EXP.I.5
failure.Hardware.N04_MonIn2.1.ACT.OMISSION
out.1.OMISSION

TRACE 7:
failure.Hardware.N04_MonIn1.1.EXP.I.5
failure.Hardware.N04_MonIn1.1.ACT.OMISSION
failure.Hardware.N04_MonIn2.1.EXP.I.5
failure.Hardware.N04_MonIn2.1.ACT.OMISSION
failure.Hardware.N04_RelationalOperator.1.EXP.B.false
failure.Hardware.N04_RelationalOperator.1.ACT.B.true

TRACE 8:
failure.Hardware.N04_MonIn2.1.EXP.I.5
failure.Hardware.N04_MonIn2.1.ACT.OMISSION
failure.Hardware.N04_MonIn1.1.EXP.I.5
failure.Hardware.N04_MonIn1.1.ACT.OMISSION
failure.Hardware.N04_RelationalOperator.1.EXP.B.false
failure.Hardware.N04_RelationalOperator.1.ACT.B.true

TRACE 9:
failure.Hardware.N04_RelationalOperator.1.EXP.B.false
failure.Hardware.N04_RelationalOperator.1.ACT.B.true
failure.Hardware.N04_MonIn1.1.EXP.I.5
failure.Hardware.N04_MonIn1.1.ACT.OMISSION
failure.Hardware.N04_MonIn2.1.EXP.I.5
failure.Hardware.N04_MonIn2.1.ACT.OMISSION

TRACE 10:
failure.Hardware.N04_RelationalOperator.1.EXP.B.false
failure.Hardware.N04_RelationalOperator.1.ACT.B.true
failure.Hardware.N04_MonIn2.1.EXP.I.5
failure.Hardware.N04_MonIn2.1.ACT.OMISSION
failure.Hardware.N04_MonIn1.1.EXP.I.5
failure.Hardware.N04_MonIn1.1.ACT.OMISSION

TRACE 11:
failure.Hardware.N04_MonIn2.1.EXP.I.5
failure.Hardware.N04_MonIn2.1.ACT.OMISSION
failure.Hardware.N04_RelationalOperator.1.EXP.B.false
failure.Hardware.N04_RelationalOperator.1.ACT.B.true
failure.Hardware.N04_MonIn1.1.EXP.I.5
failure.Hardware.N04_MonIn1.1.ACT.OMISSION
```

where `N04` is the subsystem name of the monitor in the Simulink diagram, `MonIn1` (first input of the monitor), `MonIn2` (second input of the monitor), and `RelationalOperator` (switcher controller) are the names of the hardware components in the Simulink diagram.

We only show eleven counter-examples, but FDR generates a total of 64 counter-examples for this system. The other counter-examples are similar to the traces shown with different internal events.

To reuse HiP-HOPS, which is based on SFTs, we "remove" the ordering information of the traces to generate a failure expression. Each fault trace is abstracted as a conjunction (AND combination of the inner events, thus losing the ordering information), and the several conjunction-based fault events are combined using ORs (disjunctions). The result of the combination is a Boolean expression that represents the conditions that cause an undesirable output, the failure expression of the model. With the ATF proposed in this work we do not "remove" the ordering information, so we are able to use this information to generate or perform DFT and TFT analyses (TFTs have order-related operators, and it is shown in [3, 24, 22] that DFTs can be expressed by order-related operators).

If the failure expression is obtained for a whole system, it is indeed the structure expression of a fault tree for a general failure as the top-level event. Although it is possible to obtain the failure expression for a larger system, it may be impractical due to state-space explosion in CSP_M model analysis. Thus it should be used for components and subsystems or small systems following HiP-HOPS compositional structure. Using failure expression as subsystem annotations in [25], it is possible to obtain structure expressions for a larger system. It is worth noting that the goal of the work reported in [28] was to connect with HiP-HOPS, which is based on static fault trees. But we already knew that we had a richer fault modelling information than that presented in [28] because we abstracted traces (which already capture fault events ordering) to create propositions (any fault events order combination).

To show how these traces become failure expression, let us abbreviate fault names as:

```
A = failure.Hardware.N04_MonIn1.1
B = failure.Hardware.N04_MonIn2.1
S = failure.Hardware.N04_RelationalOperator
```

Table 7 – Annotations table of the ACS provided by our industrial partner

Component	**Deviation**	**Port**	**Annotation**
PowerSource	LowPower	Out1	PowerSourceFailure
Monitor	LowPower	Out1	(SwitchFailure AND (LowPower-In1 OR LowPower-In2)) OR (LowPower-In1 AND LowPower-In2)
Reference	OmissionSignal	Out1	ReferenceDeviceFailure OR LowPower-In1

So, for each trace, we obtain an expression:

$$\begin{aligned}
\texttt{TRACE 1} &= S \wedge B \\
\texttt{TRACE 2} &= B \wedge S \\
\texttt{TRACE 3} &= A \wedge B \\
\texttt{TRACE 4} &= B \wedge A \\
\texttt{TRACE 5} &= A \wedge S \\
\texttt{TRACE 6} &= A \wedge S \wedge B \\
\texttt{TRACE 7} &= A \wedge B \wedge S \\
\texttt{TRACE 8} &= B \wedge A \wedge S \\
\texttt{TRACE 9} &= S \wedge A \wedge B \\
\texttt{TRACE 10} &= S \wedge B \wedge A \\
\texttt{TRACE 11} &= B \wedge S \wedge A
\end{aligned}$$

And we combine them as a single Boolean expression: $\texttt{TRACE 1} \vee \texttt{TRACE 2} \vee \texttt{TRACE 3} \vee \texttt{TRACE 4} \vee \texttt{TRACE 5} \vee \texttt{TRACE 6} \vee \texttt{TRACE 7} \vee \texttt{TRACE 8} \vee \texttt{TRACE 9} \vee \texttt{TRACE 10} \vee \texttt{TRACE 11}$, which by a traditional Boolean reduction strategy results in:

$$(A \wedge B) \vee (S \wedge (A \vee B))$$

The above expression is exactly the same failure expression provided by our industrial partner if we use the following association (Table 7):

$$\begin{aligned}
A &= \text{LowPower-In1} \\
B &= \text{LowPower-In2} \\
S &= \text{SwitchFailure}
\end{aligned}$$

Note that when we combine each fault with AND gates, we lose the information about order[5]: $S \wedge B$ and $B \wedge S$ are equal, due to the commutative law of Boolean expressions.

---

[5] In our previous work we designed the observer to ignore order as well, by making similar traces—with

Our strategy finds fault combinations S and B (in the sense of S occurring before B) as well as B and S (in the sense of B occurring before S) but abstracts this ordering information obtaining B and S, which is equivalent to S and B in Boolean Algebra. If A fails before S, the system fails because it should switch to B, but the switcher is in a faulty state. On the other hand, if S fails before A, the switcher fails because it inadvertently switched to B when A was still operational. When A fails, nothing changes and the output of the system is obtained from B.

We also employed the strategy proposed in the work [28] in another case study and obtained a weaker failure expression (that is, our expression considers more cases). The failure expression provided by the engineers of our industrial partner was stronger because they considered that one component has a very low probability of failure and removed it from the failure analysis. Although acceptable, it may cause incorrect analysis. Our strategy avoids this kind of issue by being completely systematic.

3.6 Isabelle/HOL

We use the same words of the creators of this tool, retrieved from their website[6]:

> Isabelle is a generic proof assistant. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus. The main application is the formalization of mathematical proofs and in particular formal verification, which includes proving the correctness of computer hardware or software and proving properties of computer languages and protocols.

Isabelle/HOL is the most widespread instance of Isabelle. HOL stands for higher-order logic. Isabelle/HOL provides a HOL proving environment ready to use, which includes: (co)datatypes, inductive definitions, recursive functions, locales, custom syntax definition, etc. Proofs can be written in both human[7] and machine-readable language based on Isar. The tool also includes the *sledgehammer*, a port to call external first-order provers to find proofs fully automatically. The user interface is based on jEdit[8], which provides a text editor, syntax parser, shortcuts, etc. (see Figure 27).

Theories on Isabelle/HOL are based on a few axioms. Isabelle/HOL Library's theories—which comes with the installer—and user's theories are based on these axioms. This design decision avoids inconsistencies and paradoxes (similar as it is in Z).

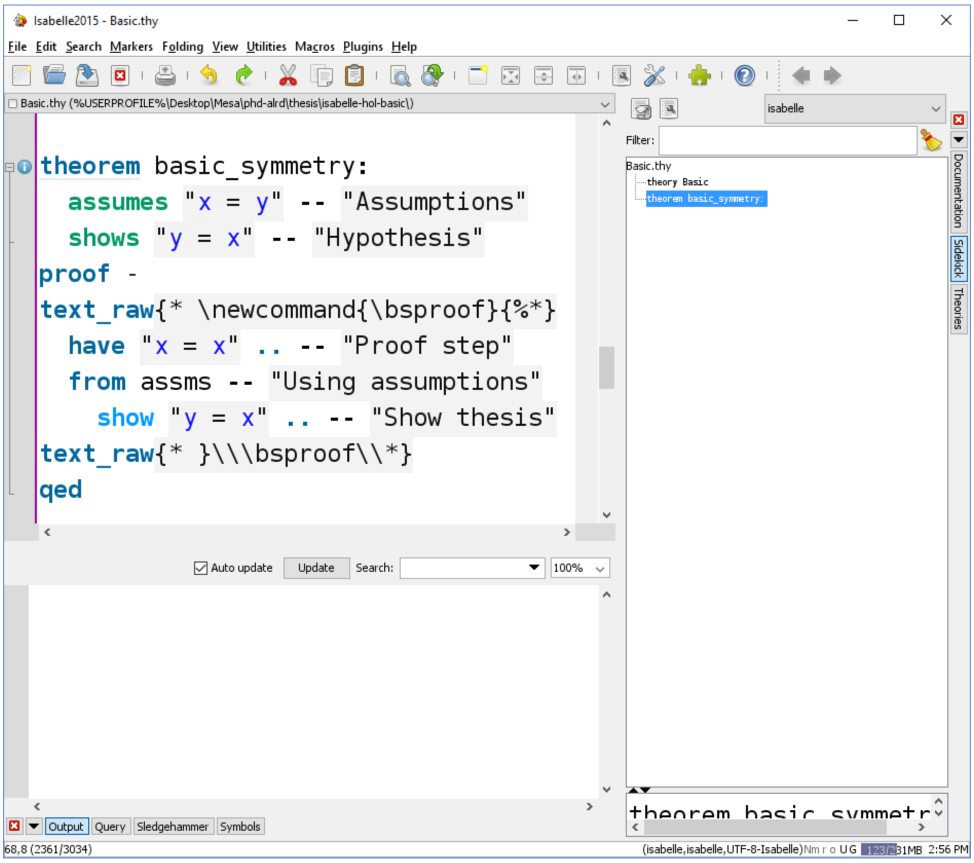
---

different ordering—the same size. Here we modified the observer specification to make similar traces with different sizes.

[6] Accessed 27/jan/2016: <https://isabelle.in.tum.de/overview.html>

[7] By human we mean that anyone with mathematics and logic basic knowledge—it means that deep programming knowledge is not essential.

[8] Accessed 27/jan/2016: <http://www.jedit.org/>


Isabelle2015 - Basic.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
Basic.thy (%USERPROFILE%\Desktop\Mesa\phd-alrd\thesis\isabelle-hol-basic\)
isabelle
Filter:
Basic.thy
theory Basic
theorem basic_symmetry:
Documentation
Sidekick
Theories
☑ Auto update
Update
Search:
100%
Output
Query
Sledgehammer
Symbols
theorem basic_symmetr
68,8 (2361/3034)
(isabelle,isabelle,UTF-8-Isabelle)Nm r o UG 123/231MB 2:56 PM

```
theorem basic_symmetry:
  assumes "x = y" -- "Assumptions"
  shows "y = x" -- "Hypothesis"
proof -
text_raw{* \newcommand{\bsproof}{%*}
  have "x = x" .. -- "Proof step"
  from assms -- "Using assumptions"
    show "y = x" .. -- "Show thesis"
text_raw{* }\\\bsproof\\*}
qed
```

Figure 27 – Isabelle/HOL window, showing the basic symmetry theorem

Besides the provided theories, its active community provides a comprehensive archive of formal proofs[9] (AFP). Each entry in this archive can be cited and usually contains an *abstract*, a document, and a theory file. For example, a Free Boolean Algebra theory is available in [85]. To use it, it is enough to download and put on the same directory of your own theory files.

Bellow we show an example and explain the overall syntax of the human and machine-readable language.

```
theorem basic_symmetry:
  assumes "x = y" — Assumptions
  shows "y = x" — Hypothesis
proof -
have "x = x" .. — Proof step
  from assms — Using assumptions
    show "y = x" .. — Show thesis
qed
```

Finally, Isabelle/HOL provides LaTeX syntax sugar and allow easy document preparation: this entire section was written in a theory file mixing Isabelle's and LaTeX's syntax). The above theorem can be written using Isabelle's quotation and anti-quotations. For example, we can write it using usual LaTeX theorem environment:

---

[9] Accessed 27/jan/2016: <http://afp.sourceforge.net/>

Theorem 3.1 (Basic symmetry)**.** *Assuming* `x = y`*, thus:*

$$y = x$$

Proof. **have** "x = x" **..** — Proof step

from assms — Using assumptions

show "y = x" **..** — Show thesis ☐

Otherwise specified, in the next sections we will omit proofs because they are all verified using Isabelle/HOL. The complete listing is in Appendix A.

4 A free algebra to express structure expressions of ordered events

Recall from Sections 2.2 and 3.1 that fault events are independent of one another if the events are not susceptible to a common cause. The set-theoretical abstraction of structure expressions for SFTs [18, pp. VI-11] is very close to an FBA, where each generator in FBAs corresponds to a fault event symbol in fault trees. In FBAs, as generators are "free", they are independent of one another and Boolean formulas are written as a set of sets of possibilities, which are similar to the structure expressions of SFTs.

We showed in Section 3.1 that there is a consistent presence of order-based operators to analyse TFTs and DFTs, and that each approach describes a new algebra based on different representations of events ordering with similar theorems to reduce expressions to a canonical form.

From the need to tackle events ordering and from the ordering information we had from fault injection that we developed in [28], we defined a list-based algebra, called Algebra of Temporal Faults (ATF), to express and analyse systems considering events ordering. We also provide a mapping from fault traces [28] (from CSP$_M$ models) to this algebra, shown in Section 4.4. The order-specific operations are expressed with a new operator ($\rightarrow$) that we call exclusive-before (XBefore).

The set of sets for FBAs are the denotational semantics for Boolean algebras. We use the concept of generators to propose the ATF with a denotational semantics of a set of lists without repetition (distinct lists[1]). The choice of lists is because this structure inherently associates a generator to an index, making implicit the representation of order. These lists are composed by non-repeated elements (distinct lists) because the events in fault trees are non-repairable, thus they do not occur more than once.

This list representation is different from the Sequence Number function used in [20, 21], but is related to the concept that there should be no gaps between consecutive events occurrence. It is different because order 0 (zero) in [20, 21] means non-occurrence. It may cause a discontinuity because 0 to 1 is different of 1 to 2. In FBAs the non-occurrence of an event is just the absence of the event. Thus we use the same representation of non-occurrence in ATF to avoid this discontinuity.

In the following we show the definitions and laws of our proposed ATF. To avoid repetition, let S, T and U be sets of distinct lists. A list xs is distinct if it has no repeated element. So, if x is in xs, then it has a unique associated index i and we denote it as $x = xs_i$, where $i \neq j$ and there is no xs_j such that $x = xs_j$. Furthermore, as we follow an FBA characterisation, we also need to show that the generators are independent.

---

[1] Although some may use the terminology "disjoint lists" to call the lists of non-repeated elements, we use the same terminology (distinct lists) of the theories built-in the Isabelle/HOL tool.

The ATF form a free algebra, similarl~~y~~ to FBAs. *Infimum* and *Supremum* are denoted as set intersection ($\cap$) and union ($\cup$) respectively. The order within the algebra is denoted with set inclusion ($\subseteq$).

To distinguish the permutations that are not defined in FBA, we need a new operator. We give the definition of XBefore ($\rightarrow$) in terms of list concatenation, similar to the work reported in [86]:

$$\begin{aligned} [\![S \rightarrow T]\!] = \{\ zs \mid \exists xs, ys \bullet (\mathbf{set}\, xs) \cap (\mathbf{set}\, ys) = \{\} \wedge \\ xs \in [\![S]\!] \wedge ys \in [\![T]\!] \wedge zs = xs \,@\, ys\ \} \end{aligned} \tag{4.1}$$

where the **set** function returns the set of the elements of a list, @ concatenates two lists, and $[\![.]\!]$ obtains the denotational semantics of the formula.

In some cases it is more intuitive to use the XBefore definition in terms of lists slicing because it uses indexes explicitly. Lists slicing is the operation of taking or dropping elements, obtaining a sublist. In slicing, the starting index is inclusive, and the ending one is exclusive. Thus the first index is 0 and the last index is the list length. For example, the list $xs_{[i..|xs|]}$ is equal to the xs list, where $|xs|$ is the list length. We use the following notation for list slicing:

$$xs_{[i..j]} = \text{starts at } i \text{ and ends at } j - 1 \tag{4.2a}$$

$$xs_{[..j]} = xs_{[0..j]} \tag{4.2b}$$

$$xs_{[i..]} = xs_{[i..|xs|]} \tag{4.2c}$$

To simplify the use of list slicing, its definition includes the lower and upper bounds as 0 and its length, respectively:

$$xs_{[i..j]} = xs_{[L..U]} \tag{4.3}$$

where $L = \max 0, i$ and $U = \min j, |xs|$.

List slicing and concatenation are complementary: concatenating two consecutive slices results in the original list:

$$\forall i \bullet xs_{[..i]} \,@\, xs_{[i..]} = xs \tag{4.4}$$

There is an equivalent definition of XBefore with concatenation using lists slicing:

$$[\![S \rightarrow T]\!] = \left\{ zs \;\middle|\; \exists i \bullet zs_{[..i]} \in [\![S]\!] \wedge zs_{[i..]} \in [\![T]\!] \right\} \tag{4.5}$$

A variable in ATF is defined by one generator, and denotes its occurrence:

$$[\![\mathbf{var}\, x]\!] = \{zs \mid x \in zs\} \tag{4.6}$$

where $set\, zs$ is the set of the elements of zs, and $x \in zs$ is defined as $x \in set\, zs$.

For example, for generators a and b, we obtain the following denotational semantics:

$$[\![\mathbf{var}\, a]\!] = \{[a], [a,b], [b,a]\} \tag{4.7a}$$

$$[\![\mathbf{var}\, b]\!] = \{[b], [a,b], [b,a]\} \tag{4.7b}$$

Given this definition, we show a small example on how the XBefore operator works:

$$\begin{aligned}
\text{For } zs = [\,],\ & [\,] \notin [\![\mathbf{var}\, a]\!] \wedge [\,] \notin [\![\mathbf{var}\, b]\!] \implies \\
& [\,] \notin [\![\mathbf{var}\, a \to \mathbf{var}\, b]\!] \\
\text{For } zs = [a],\ & [a] \in [\![\mathbf{var}\, a]\!] \wedge [a] \notin [\![\mathbf{var}\, b]\!] \wedge [\,] \notin [\![\mathbf{var}\, b]\!] \implies \\
& [a] \notin [\![\mathbf{var}\, a \to \mathbf{var}\, b]\!] \\
\text{For } zs = [b],\ & [b] \in [\![\mathbf{var}\, b]\!] \wedge [b] \notin [\![\mathbf{var}\, a]\!] \wedge [\,] \notin [\![\mathbf{var}\, a]\!] \implies \\
& [b] \notin [\![\mathbf{var}\, a \to \mathbf{var}\, b]\!] \\
\text{For } zs = [a,b],\ & [a] \in [\![\mathbf{var}\, a]\!] \wedge [b] \in [\![\mathbf{var}\, b]\!] \implies \\
& [a,b] \in [\![\mathbf{var}\, a \to \mathbf{var}\, b]\!] \\
\text{For } zs = [b,a],\ & [b] \notin [\![\mathbf{var}\, a]\!] \wedge [\,] \notin [\![\mathbf{var}\, a]\!] \wedge [b,a] \in [\![\mathbf{var}\, a]\!] \wedge \\
& [\,] \notin [\![\mathbf{var}\, b]\!] \wedge [a] \notin [\![\mathbf{var}\, b]\!] \wedge [b,a] \in [\![\mathbf{var}\, b]\!] \implies \\
& [b,a] \notin [\![\mathbf{var}\, a \to \mathbf{var}\, b]\!] \\
[\![\mathbf{var}\, a \to \mathbf{var}\, b]\!] = \{[a,b]\} &
\end{aligned} \tag{4.8}$$

Boolean operators are denoted as in FBA:

$$[\![S \wedge T]\!] = [\![S]\!] \cap [\![T]\!] \tag{4.9a}$$

$$[\![S \vee T]\!] = [\![S]\!] \cup [\![T]\!] \tag{4.9b}$$

$$[\![\neg S]\!] = -[\![S]\!] \tag{4.9c}$$

$$[\![\bot]\!] = \{\} \tag{4.9d}$$

$$[\![\top]\!] = \text{UNIV} \tag{4.9e}$$

UNIV is the universal set, that contain all permutations of the generators of size 0 to the number of the generators. For example, if the generators are a and b, then UNIV is:

$$\{\ [\,], [a], [b], [a,b], [b,a]\ \}$$

The following expressions are sufficient to define the ATF in terms of an inductively defined set (**atf**):

$$\llbracket \mathbf{var}\, x \rrbracket \in \mathbf{atf} \qquad \text{Variable} \tag{4.10a}$$

$$\llbracket S \rrbracket \in \mathbf{atf} \implies \llbracket \neg S \rrbracket \in \mathbf{atf} \qquad \text{Complement, Negation} \tag{4.10b}$$

$$\llbracket S \rrbracket \in \mathbf{atf} \wedge \llbracket T \rrbracket \in \mathbf{atf} \implies \llbracket S \wedge T \rrbracket \in \mathbf{atf} \qquad \text{Intersection, } \textit{Infimum} \tag{4.10c}$$

$$\llbracket S \rrbracket \in \mathbf{atf} \wedge \llbracket T \rrbracket \in \mathbf{atf} \implies \llbracket S \rightarrow T \rrbracket \in \mathbf{atf} \qquad \text{XBefore} \tag{4.10d}$$

Following the definitions, the expressions below are also valid for **atf**:

$$\mathrm{UNIV} \in \mathbf{atf} \qquad \text{Universal set, True} \tag{4.10e}$$

$$\{\} \in \mathbf{atf} \qquad \text{Empty set, False} \tag{4.10f}$$

$$\llbracket S \rrbracket \in \mathbf{atf} \wedge \llbracket T \rrbracket \in \mathbf{atf} \implies \llbracket S \vee T \rrbracket \in \mathbf{atf} \qquad \text{Union, } \textit{Supremum} \tag{4.10g}$$

The NOT operator is given in terms of UNIV. For example, for generators a and b:

$$\begin{aligned}
\llbracket \neg\, \mathbf{var}\, a \rrbracket &= -\, \llbracket \mathbf{var}\, a \rrbracket && \text{by Eq. (4.9c)} \\
&= -\, \{[a]\,, [a,b]\,, [b,a]\} && \text{by Eq. (4.7a)} \\
&= \{[]\,, [a]\,, [b]\,, [a,b]\,, [b,a]\} - \{[a]\,, [a,b]\,, [b,a]\} \\
&= \{[]\,, [b]\}
\end{aligned} \tag{4.11}$$

The following expressions are valid for generators a and b and are sufficient to show that the generators are independent:

$$\llbracket \mathbf{var}\, a \rrbracket = \llbracket \mathbf{var}\, b \rrbracket \iff a = b \tag{4.12a}$$

$$\llbracket \mathbf{var}\, a \rrbracket \not\subseteq -\, \llbracket \mathbf{var}\, b \rrbracket \tag{4.12b}$$

$$\llbracket \mathbf{var}\, a \rrbracket \neq -\, \llbracket \mathbf{var}\, b \rrbracket \tag{4.12c}$$

$$-\, \llbracket \mathbf{var}\, a \rrbracket \not\subseteq \llbracket \mathbf{var}\, b \rrbracket \tag{4.12d}$$

$$-\, \llbracket \mathbf{var}\, a \rrbracket \neq \llbracket \mathbf{var}\, b \rrbracket \tag{4.12e}$$

Expressions Eq. (4.10a) to Eq. (4.10g) and Eq. (4.12a) to Eq. (4.12e) implies that the ATF without the XBefore operator Eq. (4.1) forms a Boolean algebra based on sets of lists. And this is also equivalent to an FBA with the same generators.

We denote the set of generators $\mathrm{Gen}(S)$ of a formula S as:

$$\mathrm{Gen}(S) = \bigcup_{xs \in \llbracket S \rrbracket} \mathbf{set}\, xs \tag{4.13}$$

The generators of ATF are Gen(UNIV), or simply Gen.

In our previous work [86] we stated a relation of XBefore and *supremum*, provided the operands are variables Eq. (4.6). Now we generalise this relation in terms of abstract properties of the operands of the XBefore. We name these properties as *temporal properties*.

4.1 Temporal properties (*tempo*)

Temporal properties give a more abstract and less restrictive shape on the XBefore laws. They are abbreviations that some operators satisfy altogether or individually. These properties avoid the requirement that every operand of XBefore should be a variable (Eq. (4.6)).

$$\begin{aligned}
\mathbf{tempo}_1\, S &= \forall i, j, zs \bullet i \leq j \implies \neg\left(zs_{[..i]} \in [\![S]\!] \wedge zs_{[j..]} \in [\![S]\!]\right) && \text{(4.14a)}\\
\mathbf{tempo}_2\, S &= \forall i, zs \bullet zs \in [\![S]\!] \iff zs_{[..i]} \in [\![S]\!] \vee zs_{[i..]} \in [\![S]\!] && \text{(4.14b)}\\
\mathbf{tempo}_3\, S &= \forall i, j, zs \bullet j < i \implies \\
&\quad \left(zs_{[j..i]} \in [\![S]\!] \iff zs_{[..i]} \in [\![S]\!] \wedge zs_{[j..]} \in [\![S]\!]\right) && \text{(4.14c)}\\
\mathbf{tempo}_4\, S &= \forall zs \bullet zs \in [\![S]\!] \iff (\exists i \bullet zs_{[i..(i+1)]} \in [\![S]\!]) && \text{(4.14d)}
\end{aligned}$$

The first temporal property is about disjoint split. If the first part of a list is in a given set, then every remainder part is not. So, if a generator is in the beginning of a list, it must not be at the end (and vice-versa). For example, let $zs = [z_0, z_1, z_2, z_3, z_4]$. If $[z_0, z_1] \in [\![S]\!]$, thus $[z_3, z_4] \notin [\![S]\!]$. Note that it is vague, but it is the first relation indication on the lists of S. In this property, as zs has no repeated elements, then there is no element that is in both sublists.

In what follows we show how variables satisfy **tempo** properties. The denotational semantics of **var** z_1 (for generators z_1, z_2, z_3) is:

$$\begin{aligned}
[\![\mathbf{var}\ z_1]\!] = \{&[z_1], [z_1, z_2], [z_2, z_1], [z_1, z_3], [z_3, z_1], [z_1, z_2, z_3], [z_1, z_3, z_2],\\
&[z_2, z_1, z_3], [z_2, z_3, z_1], [z_3, z_1, z_2], [z_3, z_2, z_1]\} && \text{(4.15)}
\end{aligned}$$

Given the denotational, semantics, we demonstrate that the property $\mathbf{tempo}_1$ is satisfied:

$$\begin{aligned}
[z_1] \in [\![S]\!] &\implies [z_2] \notin [\![S]\!] \wedge [z_3] \notin [\![S]\!] \wedge [z_2, z_3] \notin [\![S]\!] \wedge [z_3, z_2] \notin [\![S]\!]\\
[z_1, z_2] \in [\![S]\!] &\implies [z_3] \notin [\![S]\!]\\
[z_2, z_1] \in [\![S]\!] &\implies [z_3] \notin [\![S]\!]\\
[z_1, z_3] \in [\![S]\!] &\implies [z_2] \notin [\![S]\!]\\
[z_3, z_1] \in [\![S]\!] &\implies [z_2] \notin [\![S]\!]\\
[z_1, z_2, z_3] \in [\![S]\!] &\implies [] \notin [\![S]\!]\\
&\ \ \ldots
\end{aligned}$$

The second temporal property is about belonging to one sublist in the beginning or in the end. For example, if a generator is in a list, then it must be at the beginning or at the end. If $[z_0, z_1, z_2, z_3, z_4] \in [\![S]\!]$, thus either $[z_0] \in [\![S]\!]$, or $[z_1, z_2, z_3, z_4] \in [\![S]\!]$. If

$[z_0] \notin [\![S]\!]$, thus $[z_1, z_2, z_3, z_4] \in [\![S]\!]$. Then, either $[z_1] \in [\![S]\!]$ or $[z_2, z_3, z_4] \in [\![S]\!]$, and so forth. For variable **var** z_1:

$$\begin{aligned}
[z_1] \in [\![S]\!] &\iff [z_1] \in [\![S]\!] \lor [\,] \in [\![S]\!] \\
[z_1, z_2] \in [\![S]\!] &\iff [z_1] \in [\![S]\!] \lor [z_2] \in [\![S]\!] \\
[z_1, z_2, z_3] \in [\![S]\!] &\iff ([z_1] \in [\![S]\!] \lor [z_2, z_3] \in [\![S]\!]) \land \\
& ([z_1, z_2] \in [\![S]\!] \lor [z_3] \in [\![S]\!]) \\
[z_2, z_1, z_3] \in [\![S]\!] &\iff ([z_2] \in [\![S]\!] \lor [z_1, z_3] \in [\![S]\!]) \land \\
& ([z_2, z_1] \in [\![S]\!] \lor [z_3] \in [\![S]\!]) \\
& \ldots
\end{aligned}$$

The third temporal property is about belonging to one sublist in the middle. If a generator belongs to a sublist between j and i, then it belongs to the sublist that starts at first position and ends in j and to the sublist that starts at i and ends at the last position (both sublists contain the sublist in the middle). For example, if $[z_1, z_2, z_3] \in [\![S]\!]$, then both $[z_0, z_1, z_2, z_3]$ and $[z_1, z_2, z_3, z_4] \in [\![S]\!]$.

$$\begin{aligned}
[z_1] \in [\![S]\!] &\iff [z_1] \in [\![S]\!] \land [z_1] \in [\![S]\!] \land \\
& [z_2, z_1] \in [\![S]\!] \land [z_1] \in [\![S]\!] \land \\
& [z_3, z_1] \in [\![S]\!] \land [z_1] \in [\![S]\!] \land \\
& [z_1] \in [\![S]\!] \land [z_1, z_2] \in [\![S]\!] \land \\
& [z_1] \in [\![S]\!] \land [z_1, z_3] \in [\![S]\!] \land \\
& [z_2, z_1] \in [\![S]\!] \land [z_1, z_3] \in [\![S]\!] \land \\
& [z_3, z_1] \in [\![S]\!] \land [z_1, z_2] \in [\![S]\!] \\
[z_1, z_2] \in [\![S]\!] &\iff [z_3, z_1, z_2] \in [\![S]\!] \land [z_1, z_2] \in [\![S]\!] \\
& [z_1, z_2] \in [\![S]\!] \land [z_1, z_2, z_3] \in [\![S]\!] \\
& \ldots
\end{aligned}$$

Finally, if a generator belongs to a list, then there is a sublist of size one that contains the generator. If list $zs = [z_0, z_1, z_2, z_3, z_4] \in [\![S]\!]$, then one list $[z_i] \in [\![S]\!]$, where $i \in \{0, \ldots, 4\}$. For a variable **var** z_1

$$\begin{aligned}
[z_1] \in [\![S]\!] &\iff [z_1] \in [\![S]\!] \\
[z_1, z_2] \in [\![S]\!] &\iff [z_1] \in [\![S]\!] \lor [z_2] \in [\![S]\!] \\
[z_1, z_2, z_3] \in [\![S]\!] &\iff [z_1] \in [\![S]\!] \lor [z_2] \in [\![S]\!] \lor [z_3] \in [\![S]\!] \\
& \ldots
\end{aligned}$$

In general, for any generator z, the following is valid:

$$\mathbf{tempo}_1\,(\mathbf{var}\ z) \land \mathbf{tempo}_2\,(\mathbf{var}\ z) \land \mathbf{tempo}_3\,(\mathbf{var}\ z) \land \mathbf{tempo}_4\,(\mathbf{var}\ z) \tag{4.16}$$

In our previous work [86] we used set difference to specify the XBefore operator. Provided $\mathbf{tempo}_1\, S$ and $\mathbf{tempo}_1\, T$, XBefore in [86] is equivalent to Eq. (4.1):

$$\llbracket S \rightarrow T \rrbracket = \{zs \mid \exists xs, ys \bullet xs \in \llbracket S \rrbracket - \llbracket T \rrbracket \wedge ys \in \llbracket T \rrbracket - \llbracket S \rrbracket \wedge \\ \text{distinct}\, zs \wedge zs = xs \mathbin{@} ys\} \tag{4.17}$$

Other expressions also meet one or more temporal properties:

$$\mathbf{tempo}_1\, S \wedge \mathbf{tempo}_1\, T \implies \mathbf{tempo}_1\, (S \wedge T) \tag{4.18a}$$
$$\mathbf{tempo}_3\, S \wedge \mathbf{tempo}_3\, T \implies \mathbf{tempo}_3\, (S \wedge T) \tag{4.18b}$$
$$\mathbf{tempo}_2\, S \wedge \mathbf{tempo}_2\, T \implies \mathbf{tempo}_2\, (S \vee T) \tag{4.18c}$$
$$\mathbf{tempo}_4\, S \wedge \mathbf{tempo}_4\, T \implies \mathbf{tempo}_4\, (S \vee T) \tag{4.18d}$$

4.2 XBefore laws

We now show some laws to be used in the algebraic reduction of ATF formulas. The laws follow from the definition of XBefore, from events independence, and from the temporal properties.

We use a normal form similar to the DNF of Boolean algebra. In DNF each sub-expression is a minimal cut set for SFT. In our normal form, also called DNF, we allow ANDs, NOTs, and XBefores to be in the sub-expressions. Each sub-expression is a set of minimal cut sequences for TFT and DFT. The following formulas are in DNF:

$$(A \wedge -B) \vee ((A \rightarrow B) \wedge C)$$
$$A \vee B$$
$$A \rightarrow B$$
$$A \wedge B$$
$$A \rightarrow B \rightarrow C$$

The following formulas are *not* in DNF:

$$-(A \vee B)$$
$$A \wedge (B \vee C)$$
$$A \rightarrow (B \vee C)$$
$$A \rightarrow (B \wedge C)$$

But to transform the last two formulas into DNF, one can use Eqs. (4.23a) to (4.23d), for instance.

We define events independence ($\lhd\!\rhd$) as the property that one operand does not imply the other. For example, we need to avoid that the operands of XBefore are $\mathbf{var}\,a$ and $\mathbf{var}\,a \vee \mathbf{var}\,b$ (it results in $\{\}$, see Eq. (4.20g)).

$$S \lhd\!\rhd T = \forall i, zs \bullet \neg\left(zs_{[i..(i+1)]} \in [\![S]\!] \wedge zs_{[i..(i+1)]} \in [\![T]\!]\right) \tag{4.19}$$

For generators a and b and the denotational semantics of $\mathbf{var}\,a$ and $\mathbf{var}\,b$ (see Eqs. (4.7a) and (4.7b)), it is easy to check that $\mathbf{var}\,a \lhd\!\rhd \mathbf{var}\,b$ is satisfied, and $\mathbf{var}\,a \lhd\!\rhd (\mathbf{var}\,a \vee \mathbf{var}\,b)$ is not. Essentially if the formulas contain independent events one must not imply on the other on the occurrence of individual events ($zs_{[i..(i+1)]}$).

The absence of occurrences ($\{\}$, the empty set of **atf**) is a "0" for the XBefore operator. The negation of all events ($\{[]\}$, a formula with the empty list) is a "1" for the XBefore operator.

$$\begin{aligned}
\bot \rightarrow S &= \bot & \text{(4.20a)}\\
S \rightarrow \bot &= \bot & \text{(4.20b)}\\
\mathbf{tempo}_1\, S \implies \mathbf{1} \rightarrow S &= S & \text{(4.20c)}\\
\mathbf{tempo}_1\, S \implies S \rightarrow \mathbf{1} &= S & \text{(4.20d)}\\
(S \rightarrow T) \vee S &= S & \text{(4.20e)}\\
(T \rightarrow S) \vee S &= S & \text{(4.20f)}\\
\mathbf{tempo}_1\, S \implies S \rightarrow S &= \bot & \text{(4.20g)}\\
\mathbf{tempo}_1\, S \wedge \mathbf{tempo}_1\, T \wedge \mathbf{tempo}_1\, U &\implies & \\
S \rightarrow (T \rightarrow U) &= (S \rightarrow T) \rightarrow U & \text{(4.20h)}
\end{aligned}$$

The XBefore is absorbed by one of the operands: if one of the operands may happen alone, thus the order with any other operand is irrelevant. However, an event cannot come before itself, thus XBefore is not idempotent. The XBefore is associative.

To allow formula reduction we need the relation of XBefore to the other Boolean operators. First we use the XBefore as operands of OR and AND.

$$\begin{aligned}
&\mathbf{tempo}_1\, S \wedge \mathbf{tempo}_1\, T \implies \\
&(S \rightarrow T) \wedge (T \rightarrow S) = \bot & \text{(4.21a)}\\
&\mathbf{tempo}_{1-4}\, S \wedge \mathbf{tempo}_{1-4}\, T \wedge S \lhd\!\rhd T \implies \\
&(S \rightarrow T) \vee (T \rightarrow S) = S \wedge T & \text{(4.21b)}
\end{aligned}$$

As XBefore is not symmetric, the intersection of symmetrical XBefores is empty. The OR of symmetrical XBefores is a partition of the intersection of the operands. For example, given generators a and b ($a \neq b$):

$$(\mathbf{var}\,a \rightarrow \mathbf{var}\,b) \vee (\mathbf{var}\,b \rightarrow \mathbf{var}\,a) = \mathbf{var}\,a \wedge \mathbf{var}\,b \tag{4.22}$$

because $\mathbf{var}\, a$ and $\mathbf{var}\, b$ satisfy all temporal properties (Eq. (4.16)) and are independent events.

In our previous work [86], we stated that S and T had to be variables, for example, of the form $\mathbf{var}\, s$ and $\mathbf{var}\, t$. Now, each law requires that the operands satisfy some of the temporal properties, avoiding using variables explicitly.

Expressions with Boolean operators are used as operands of the XBefore in the following laws.

$$
\begin{aligned}
(S \lor T) \rightarrow U &= (S \rightarrow U) \lor (T \rightarrow U) && \text{(4.23a)}\\
S \rightarrow (T \lor U) &= (S \rightarrow T) \lor (S \rightarrow U) && \text{(4.23b)}\\
\mathbf{tempo}_{1-4}\, S \land \mathbf{tempo}_{1-4}\, T \land S \lhd\!\rhd T \implies & \\
(S \land T) \rightarrow U &= (S \rightarrow T \rightarrow U) \lor \\
& \quad (T \rightarrow S \rightarrow U) && \text{(4.23c)}\\
\mathbf{tempo}_{1-4}\, T \land \mathbf{tempo}_{1-4}\, U \land T \lhd\!\rhd U \implies & \\
S \rightarrow (T \land U) &= (S \rightarrow T \rightarrow U) \lor \\
& \quad (S \rightarrow U \rightarrow T) && \text{(4.23d)}\\
\mathbf{tempo}_{2}\, S \implies S \land (T \rightarrow U) &= ((S \land T) \rightarrow U) \lor \\
& \quad (T \rightarrow (S \land U)) && \text{(4.23e)}\\
\mathbf{tempo}_{1}\, T \land \mathbf{tempo}_{3}\, T \implies & \\
S \rightarrow T \land T \rightarrow U &= (S \rightarrow T) \rightarrow U && \text{(4.23f)}\\
S \lhd\!\rhd T \land \mathbf{tempo}_{1-4}\, S \land \mathbf{tempo}_{1-4}\, T \implies & \\
S \land (S \rightarrow T) &= S \rightarrow T && \text{(4.23g)}\\
S \lhd\!\rhd T \land \mathbf{tempo}_{1-4}\, S \land \mathbf{tempo}_{1-4}\, T \implies & \\
S \land (T \rightarrow S) &= T \rightarrow S && \text{(4.23h)}
\end{aligned}
$$

XBefore is distributive over OR. On the other hand, the AND is not. The AND of an event with an XBefore states that such an event can occur in any order within the events in the XBefore. Lastly, the XBefore is transitive with preconditions over the intermediary variable.

The law name, unordered, of Eq. (4.23e) is clearer if we expand Eq. (4.23e) with Eq. (4.23c) and Eq. (4.23d):

$$
\begin{aligned}
&\mathbf{tempo}_{1-4}\, S \land \mathbf{tempo}_{1-4}\, T \land \\
&\mathbf{tempo}_{1-4}\, U \land S \lhd\!\rhd T \land S \lhd\!\rhd U \implies \\
&\qquad S \land (T \rightarrow U) = (S \rightarrow T \rightarrow U) \lor \\
&\qquad\qquad (T \rightarrow S \rightarrow U) \lor \\
&\qquad\qquad (T \rightarrow U \rightarrow S) \qquad \text{expanded-unordered} \qquad \text{(4.24)}
\end{aligned}
$$

In what follows we show properties of XBefore related to the NOT operators.

$$
\begin{aligned}
S \lhd\!\rhd T \land \mathbf{tempo}_{1-4}\, S \land \mathbf{tempo}_{1-4}\, T \implies& \\
\neg\,(S \rightarrow T) = (\neg S \lor \neg T) \lor (T \rightarrow S)& \qquad \text{(4.25a)}\\
\mathbf{tempo}_{1}\, S \land \mathbf{tempo}_{2}\, T \implies& \\
\neg S \rightarrow T = T& \qquad \text{(4.25b)}\\
\mathbf{tempo}_{1}\, T \land \mathbf{tempo}_{2}\, S \implies& \\
S \rightarrow \neg T = S& \qquad \text{(4.25c)}\\
S \lhd\!\rhd T \land \mathbf{tempo}_{1-4}\, S \land \mathbf{tempo}_{1-4}\, T \implies& \\
(S \land \neg T) \lor (S \rightarrow T) = S \land \neg\,(T \rightarrow S)& \qquad \text{(4.25d)}
\end{aligned}
$$

4.3 Qualitative and quantitative properties

In Section 3.1 we showed the kind of results that are obtained in FTA. In this section we show how to formalize two kinds of properties of FTs: (i) MCSeq, the number of fault elements in the minimal sequences that cause a root failure, and (ii) root probability, given the availability of basic fault occurrences probabilities. These properties are the most representative, but other criteria can be modelled similarly.

Recall from the beginning of this chapter that the denotational semantics of ATF is a set of distinct lists. Thus each list has no repeated elements and represent a possible combination of faults that causes the root failure. MCSeqs are those distinct lists with the least length, defined in what follows.

Definition 4.1 (Minimal cut sequences)**.** *Let S be a formula in ATF ~~and its denotational semantics ($[\![S]\!]$)~~. The minimal cut sequences* MCSeq *of S are:*

$$
\mathrm{MCSeq}\,(S) = \{\ xs \mid xs \in [\![S]\!] \ \bullet\ |xs| = min_S\ \} \tag{4.26}
$$

where

$$
min_S = \mathrm{Min}\,(\{\ |xs| \mid xs \in [\![S]\!]\ \})
$$

and Min *returns the smallest number in the given set.*

The work reported in [3] shows how to calculate the probability of a PAND gate as:

$$
\begin{aligned}
P\,(t) &= \Pr\{T_1 \le T_2\}^{<t}\\
&= \int_0^t P_2'\boxed{(x)P_1(x)dx}
\end{aligned}
\tag{4.27}
$$

where P_1 and P_2 are the probability of the occurrence of the the first and the second faults (cumulative distribution function), respectively, and T_1 and T_2 are the times the first and the second faults occur. The general case (reported in [87]) of a PAND gate with n inputs is the probability of the sequence of faults f_i, $i \in \{ 1, \ldots, n \}$:

$$\Pr\{[f_1, f_2 \ldots, f_{n-1}, f_n]\} = \int_0^t P'_n(t_n) \int_0^{t_n} P'_{n-1}(t_{n-1}) \ldots \int_0^{t_3} P'_2(t_2) \int_0^{t_2} P'_1(t_1)\, dt_1 dt_2 \ldots dt_{n-1} dt_n \tag{4.28}$$

where P_i is the cumulative probability function of fault f_i.

The probability of occurrence in each list in the denotational semantics of ATF uses the probability in the PAND gate. The difference from the PAND calculation is that repeated situations must be removed. For example, the probability of $[\![\textbf{var}\, f_1 \rightarrow \textbf{var}\, f_2]\!]$ contains the situations that f_1 occurs before f_2 *and* f_3 does not occur, or f_3 occurs in some time: $\Pr\{[f_1, f_2, f_3]\}$, $\Pr\{[f_1, f_3, f_2]\}$, and $\Pr\{[f_3, f_1, f_2]\}$.

Using Eqs. (4.13) and (4.28) we define the ~~denotational~~ probability DPr of a list as:

$$\mathrm{DPr}\{xs\} = \Pr\{xs\} \times \prod_{f_j \in \mathrm{Gen}_{xs}} (1 - P_j(t)) \tag{4.29a}$$

$$\mathrm{Gen}_{xs} = \mathrm{Gen}(\mathrm{UNIV}) - \textbf{set}\, xs \tag{4.29b}$$

For example, for generators f_1, f_2, and f_3, $\mathrm{DPr}\{[f_1, f_2]\} = \Pr\{[f_1, f_2]\} \times (1 - P_3(t))$, which is the probability of f_1 occurring before f_2 and f_3 not occurring.

The set of lists of the denotational semantics of a formula is an exclusive or, so two possibilities (lists) are not active simultaneously. For example, if f_1 occurs before f_2, then f_2 cannot occur before f_1. Thus the formula is the sum of the denotational probabilities of the lists.

Definition 4.2 (Probability of a formula in ATF)**.** *Let S be a formula in ATF, then the probability of S,* $\mathrm{FPr}\{S\}$*, is:*

$$\mathrm{FPr}\{S\} = \sum_{xs \in [\![S]\!]} \mathrm{DPr}\{xs\} \tag{4.30}$$

The interesting part behind the probabilistic calculus over the denotational semantics is that it is only about ordering of events. It means that even if a formula contains a NOT operator, we still safely obtain a probability value, without concerning about complement probabilities as tackled in [9]. For example, the probability of $\neg\, \textbf{var}\, f_1$ (for generators f_1 and f_2) is:

$$\begin{aligned}
\Pr\{\neg\,\textbf{var}\,f_1\} =& \mathrm{DPr}\{[]\} + \mathrm{DPr}\{[f_2]\} && \text{by Eq. (4.30)}\\
=& \Pr\{[]\} \times (1 - P_1(t)) \times (1 - P_2(t))\\
&+ \Pr\{[f_2]\} \times (1 - P_1(t)) && \text{by Eq. (4.29a)}\\
=& 1 \times (1 - P_1(t)) \times (1 - P_2(t))\\
&+ P_2(t) \times (1 - P_1(t)) && \text{by Eq. (4.28)}\\
=& 1 - P_1(t) && (4.31)
\end{aligned}$$

The empty list is the special and has value 1. It works as the universal complement probability of any other list. When the empty list appears in a denotational semantics it means that the top-event occurs if no fault occurs.

We use the traditional probability calculations [3] as reference to calculate the probabilities of formulas in ATF.

$$\Pr\{\textbf{var}\,f_1 \wedge \textbf{var}\,f_2\} = P_1(t) \times P_2(t) \qquad (4.32a)$$

$$\Pr\{\textbf{var}\,f_1 \vee \textbf{var}\,f_2\} = P_1(t) + P_2(t) - P_1(t) \times P_2(t) \qquad (4.32b)$$

For example, the formula $(\textbf{var}\,f_1 \rightarrow \textbf{var}\,f_2) \vee (\textbf{var}\,f_2 \rightarrow \textbf{var}\,f_1)$, considering only the two generators (f_1 and f_2), has denotational semantics $\{[f_1, f_2], [f_2, f_1]\}$ and the probability of the formula is the probability of $[f_1, f_2]$ *or* $[f_2, f_1]$ (but not both):

$$\begin{aligned}
&\mathrm{FPr}\{(\textbf{var}\,f_1 \rightarrow \textbf{var}\,f_2) \vee (\textbf{var}\,f_2 \rightarrow \textbf{var}\,f_1)\}\\
&= \mathrm{DPr}\{[f_1, f_2]\} + \mathrm{DPr}\{[f_2, f_1]\} && \text{by Eq. (4.30)}\\
&= \Pr\{[f_1, f_2]\} + \Pr\{[f_2, f_1]\} && \text{Eq. (4.29a), both } \textstyle\sum \text{ is } 0\\
&= \int_0^t P_2'(x)P_1(x)dx + \int_0^t P_1'(x)P_2(x)dx && \text{by Eq. (4.27)}\\
&= \int_0^t (P_2'(x)P_1(x) + P_1'(x)P_2(x))\,dx && \text{by sum of } \textstyle\int\\
&= \int_0^t (P_1(x)P_2(x))'\,dx && \text{by inv. deriv. product}\\
&= P_1(t) \times P_2(t) && (4.33)
\end{aligned}$$

In Eq. (4.33) we demonstrated that the probability of a formula $(\textbf{var}\,f_1 \rightarrow \textbf{var}\,f_2) \vee (\textbf{var}\,f_2 \rightarrow \textbf{var}\,f_1)$ is equal to the probability of the traditional AND probability $(\textbf{var}\,f_1 \wedge \textbf{var}\,f_2)$, shown in Eq. (4.32a). This is expected as these two formulas are equivalent, as shown in Eq. (4.21b).

Another example including complement is the formula $\textbf{var}\ f_1 \wedge \textbf{var}\ f_2 \wedge \neg\ \textbf{var}\ f_3$:

$$\begin{aligned}
S =& \textbf{var}\ f_1 \wedge \textbf{var}\ f_2 \wedge \neg\ \textbf{var}\ f_3 && \text{by Eq. (4.30)}\\
\Pr\{S\} =& \text{DPr}\{[f_1, f_2]\} + \text{DPr}\{[f_2, f_1]\} && \text{by Eq. (4.29a)}\\
=& \Pr\{[f_1, f_2]\} \times (1 - P_3(t)) + \Pr\{[f_2, f_1]\} \times (1 - P_3(t))\\
=& P_1(t) \times P_2(t) \times (1 - P_3(t)) && \text{by Eq. (4.33)} \qquad (4.34)
\end{aligned}$$

Finally, using Eq. (4.30), and for generators f_1 and f_2, we demonstrate the equivalence to Eq. (4.32b) (the probability of an OR operator), calculated using the denotational semantics:

$$\begin{aligned}
S =& \textbf{var}\ f_1 \vee \textbf{var}\ f_2\\
\text{FPr}\{S\} =& \text{DPr}\{[f_1]\} + \text{DPr}\{[f_2]\} +\\
& \text{DPr}\{[f_1, f_2]\} + \text{DPr}\{[f_2, f_1]\} && \text{by Eq. (4.30)}\\
=& \Pr\{[f_1]\} \times (1 - P_2(t)) + \Pr\{[f_2]\} \times (1 - P_1(t)) +\\
& P_1(t) \times P_2(t) && \text{by Eqs. (4.29a) and (4.33)}\\
=& P_1(t) \times (1 - P_2(t)) + P_2(t) \times (1 - P_1) +\\
& P_1(t) \times P_2(t)\\
=& P_1(t) + P_2(t) - P_1(t) \times P_2(t) && (4.35)
\end{aligned}$$

4.3.1 Formal acceptance criteria

To enable the formal verification of structure expressions we use the concept of *acceptance criteria.* Safety requirements are written in terms of the properties of an FT, for example: (i) no single failure should cause a critical failure (the length the MCSeqs should be greater than 1), or (ii) the probability of all critical failure should be less than P_x. To check these requirements we translate these requirements into a value and verify in the theorem prover.

For the two properties shown in this section, we define the two acceptance criteria:

$$\begin{aligned}
|F|^{\leq n} =& \text{Min}(\{\ |xs| \mid xs \in L\ \}) \leq n && \text{length of MCSeqs} \qquad (4.36a)\\
\Pr\{F\}^{<P_x} =& \Pr\{F\} < P_x && \text{root-event probability} \qquad (4.36b)
\end{aligned}$$

Both equations have Boolean return value, which can be verified by a theorem prover.

4.4 Mapping CSP$_M$ traces to ATF

In our previous work [28, 27] we reported a strategy to inject faults in a CSP$_M$ model generated from an architectural model of a system in Simulink. Using a model-checker we are able to obtain failure traces with the injected faults, which we produce failure expression in Boolean algebra. Thus, the order-related information was lost, as there are no means to represent them in Boolean algebra. For example, a trace that represent f_1 before f_2, $[\ f_1, f_2\ ]$, would be written as the expression $\mathbf{var}\ f_1 \wedge \mathbf{var}\ f_2$, which is the same as $\mathbf{var}\ f_2 \wedge \mathbf{var}\ f_1$.

Using the same strategy, but now mapping the traces to ATF we are able to analyse order-related failures. In this section we show how to map the same failure traces reported in the strategy of [28] to ATF.

The mapping function takes the set of traces as input and produces a structure expression in ATF. Each event in the traces becomes a variable, and each list becomes a XBefore with the conjunction (ANDs) of the variables of the remaining generators. The following recursive definitions describe the mapping function.

$$\langle[\ \ ]\rangle_{XB} = \top \tag{4.37a}$$

$$\langle[\ f\ ]\rangle_{XB} = \mathbf{var}\ f \tag{4.37b}$$

$$\langle[\ f_1\ ] @ tr\rangle_{XB} = \mathbf{var}\ f_1 \rightarrow \langle tr\rangle_{XB} \tag{4.37c}$$

$$\langle\{tr_1, tr_2, \ldots, tr_n\}\rangle_{XB} = \bigvee_{i\in\{1,\ldots,n\}} \left(\langle tr_i\rangle_{XB} \wedge \neg \bigvee_{j\in \mathrm{Gen}_{tr_i}} \mathbf{var}\ f_j \right) \tag{4.37d}$$

where @ concatenates two traces, and Gen_{tr_i} is the same as in Eq. (4.29b) for traces. The following examples shows how it works for generators f_1, f_2, f_3:

$$\langle[\ f_1\ ]\rangle_{XB} = \mathbf{var}\ f_1 \wedge \neg(\mathbf{var}\ f_2 \vee \mathbf{var}\ f_3) \tag{4.38a}$$

$$\langle[\ f_1, f_2\ ]\rangle_{XB} = \mathbf{var}\ f_1 \rightarrow \mathbf{var}\ f_2 \wedge \neg\,\mathbf{var}\ f_3 \tag{4.38b}$$

$$\langle[\ f_3, f_2, f_1\ ]\rangle_{XB} = \mathbf{var}\ f_3 \rightarrow (\mathbf{var}\ f_2 \rightarrow \mathbf{var}\ f_1) \tag{4.38c}$$

$$\begin{aligned}\langle\{[\ f_1\ ], [\ f_1, f_2\ ]\}\rangle_{XB} = {} & (\mathbf{var}\ f_1 \wedge \neg\,\mathbf{var}\ f_2 \wedge \neg\,\mathbf{var}\ f_3) \vee \\ & ((\mathbf{var}\ f_1 \rightarrow \mathbf{var}\ f_2) \wedge \neg\,\mathbf{var}\ f_3)\end{aligned} \tag{4.38d}$$

5 Reasoning fault activation

The AL proposed in this work emerges from the need to analyse the behaviour of a system when a subset of the faults is active during the same time period, and to provide completeness analysis of system behaviour. There are at least two strategies to use AL to obtain structure expressions of SFT, TFT, or DFT: (i) model systems directly in AL, and (ii) obtaining operational mode expressions extracted from failure traces, as shown in the work reported in [52]. In approaches as those reported in [20, 3], behavioural completeness is left for the analyst. Using tautology and the indication of undefined nominal values, we ensure that no situation is left forgotten.

The AL associates: (i) an operational mode, and (ii) the expression of fault events that *activates* the operational mode or error event. The expressions of fault events can be written in any algebra that provides tautology and contradiction properties. Thus, AL is parametrized by: (i) an algebra that provides at least tautology and contradiction, and (ii) operational modes. Figure 28 depicts an overview of AL.

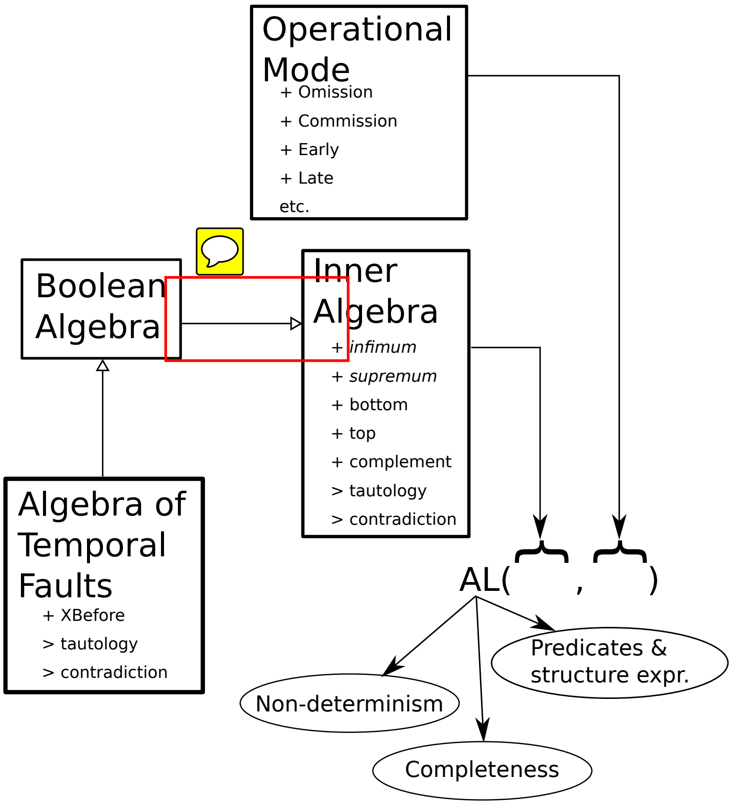

Operational Mode
+ Omission
+ Commission
+ Early
+ Late
etc.
Boolean Algebra
Inner Algebra
+ infimum
+ supremum
+ bottom
+ top
+ complement
> tautology
> contradiction
Algebra of Temporal Faults
+ XBefore
> tautology
> contradiction
AL(,)
Predicates & structure expr.
Non-determinism
Completeness

Figure 28 – AL overview

We summarise the properties of the AL as follows:

1. No expression predicate is a contradiction: there are no *false* predicates in activation

expressions;

2. The predicates in the terms of an expression consider all possible situations: expression tautology;

3. There are no two terms with exactly the same operational mode: all expression terms are related to a unique operational mode.

These properties form the *healthiness conditions* [88] of an expression in the AL.

We show the general form of the AL to model faults in Section 5.1, the healthiness conditions to normalize expressions in Section 5.2, how to identify non-determinism in an expression in Section 5.3, and the predicates notation to analyse systems and model fault propagation in Section 5.4.

5.1 The Activation Logic Grammar

Each term in an expression is a pair of a predicate and an operational mode. The predicate is written in either Boolean algebra, ATF, or any algebra that provides these properties: tautology and contradiction. We assume that the set of possible faults on a system is finite and that each variable declared in a predicate represents a fault event.

The operational mode has two generic values: (i) Nominal, and (ii) Failure. Nominal values either determined value, or an undefined value (in this case, the constant value "*undefined*" is assumed). Failure values denote an erroneous behaviour, which can be a total failure (for example, signal omission) or a failure that causes degradation (for example, a signal below or above its nominal range). The choice of the operational modes depends on the system being analysed and its definition is generic and is left for the analyst. For the AL, it is sufficient to specify that it is an erroneous behaviour.

The grammar is parametrized by the syntax of an algebra (`Algebra`) and a set of operational modes (`OperModes`). The initial rules of the grammar are defined as follows:

```
AL(Algebra, OperModes)      = TERM(Algebra, OperModes)
                              | TERM(Algebra, OperModes)
                                ‘|’ AL(Algebra, OperModes)
TERM(Algebra, OperModes)    = ‘(’ Algebra ‘,’ OM(OperModes) ‘)’
OM(OperModes)               = ‘Nominal’ NominalValue
                              | ‘Failure’ OperModes
NominalValue                = ‘undefined’ | Number
Number                      = Integer | Bool | Decimal
```

The denotational semantics of the expressions in AL is a set of pairs. The predicate in each term of an expression depends on the semantics of the inner algebra. Thus the predicate evaluates to either *true* ($\top$) or *false* ($\bot$) depending on the valuation in the algebra. In what follows we show a sketch of the denotational semantics of AL.

$$
\begin{aligned}
(\mathtt{P}_1, \mathtt{O}_1) &\mapsto \{(P_1, O_1)\} \\
(\mathtt{P}_1, \mathtt{O}_1) \,|\, (\mathtt{P}_2, \mathtt{O}_2) &\mapsto \{(P_1, O_1), (P_2, O_2)\} \\
\mathtt{Nominal\ 100} &\mapsto \mathrm{Nominal}\, 100 \\
\mathtt{Nominal\ undefined} &\mapsto \mathrm{Nominal}\, undefined \\
\mathtt{Failure\ Omission} &\mapsto \mathrm{Failure}\, Omission
\end{aligned}
$$

In an expression, if the ith predicate evaluates to *true* ($\top$), we say that the ith operational mode is *activated*. To simplify the presentation of the expressions and to ease the understanding, we use the denotational semantics in the remainder of this article (the right-hand side of the sketch above).

In this section, to illustrate the properties and possible analyses, we use an example of a system with faults A and B and the following outputs:

O_1: when A is active;

O_2: when B is active;

O_3: when A is active, but B is not;

O_4: when A or B are active.

The expression for this example in AL is:

$$S = \{(A, O_1), (B, O_2), (A \land \neg B, O_3), (A \lor B, O_4)\} \tag{5.1}$$

In this example we see that one of the healthiness conditions is not satisfied: when for instance, A and B are both inactive ($\neg(A \land B)$), there is no explicit output defined. In **??** we show a more detailed case study to illustrate the reasoning about temporal faults. In the next section, we show how to normalise the expression, so that the three healthiness conditions are satisfied.

5.2 Healthiness Conditions

The healthiness conditions are fix points of a language. The property is defined as a function of an expression and returns another expression. For example, if a healthiness condition H is satisfied for an expression *Exp*, thus $\mathrm{H}(Exp) = Exp$.

In what follows we show the three healthiness conditions for the AL. All definitions in this section refer to an algebra that has the following properties:

contradiction: the expression always evaluates to *false*;

tautology: the expression always evaluates to *true*.

5.2.1 H_1: No predicate is a contradiction.

This property is very simple and it is used to eliminate any term that has a predicate that always evaluates to false.

Definition 5.1. *Let exp be an expression in the AL, then:*

$$\mathrm{H}_1\left(exp\right) = \left\{\left(P,O\right) \mid \left(P,O\right) \in exp \bullet \neg \mathrm{contradiction}\left(P\right)\right\} \tag{5.2}$$

where the operator $\in$ indicates that a term is present in the expression.

Applying the first healthiness condition to our example results in:

$$\mathrm{H}_1\left(S\right) = S$$

Thus, we conclude that S *is* H_1-healthy.

5.2.2 H_2: All possibilities are covered.

This property is used to make explicit that there are uncovered operational modes. In this case, there is a combination of variables in the inner algebra that was not declared in the expression. Very often the focus when modelling faults is the erroneous behaviour, so we assume that such an uncovered operational mode is nominal, but has an undefined value.

Definition 5.2. *Let exp be an expression in the AL, and τ is:*

$$\tau = \neg\left(\bigvee_{(P,O)\in exp} P\right)$$

then:

$$\mathrm{H}_2\left(exp\right) = \begin{cases} exp, & \textit{if}\ \mathrm{contradiction}\left(\tau\right) \\ exp \cup \left\{\left(\tau, \mathrm{Nominal}\, undefined\right)\right\}, & \textit{otherwise} \end{cases} \tag{5.3}$$

This property states that if the expression is already complete, so all possibilities are already covered, thus the expression is healthy.

Applying the second healthiness condition to our example results in the following expression after simplification:

$$\mathrm{H}_2\left(S\right) = S \cup \left\{\left(\neg A \wedge \neg B, \mathrm{Nominal}\, undefined\right)\right\}$$

Thus, we conclude that S *is not* H_2-healthy.

5.2.3 H_3: There are no two terms with exactly the same operational mode.

This property merges terms that contain the same operational mode. It avoids unnecessary formulas and may reduce the expression.

Definition 5.3. *Let exp be an expression in the AL. Then:*

$$\begin{aligned} \mathrm{H}_3\left(exp\right) = \{ \ & \left(P_1, O_1\right) \mid \left(P_1, O_1\right) \in exp \wedge \\ & \forall \left(P_2, O_2\right) \in exp \bullet \left(P_1, O_1\right) = \left(P_2, O_2\right) \vee O_1 \neq O_2 \ \} \cup \\ & \left\{\left(P_1 \vee P_2, O_1\right) \mid \left(P_1, O_1\right), \left(P_2, O_2\right) \in exp \wedge O_1 = O_2\right\} \end{aligned} \tag{5.4}$$

Applying H_3 in the example in the beginning of the chapter, we conclude that S *is* H_3-healthy. On the other hand, if we consider an S' system being a copy of S, but making $O_1 = O_2$, then:

$$\mathrm{H}_3\left(S'\right) = \left\{\left(A \vee B, O_1\right), \left(A \wedge \neg B, O_3\right), \left(A \vee B, O_4\right)\right\}$$

Thus, we conclude that S' *is not* H_3-healthy.

5.2.4 Healthy expression

To obtain a healthy expression, we apply all three healthiness conditions. The order of application of each healthiness condition does not change the resulting expression. The healthiness function is written as composition of functions as follows:

$$\mathrm{H} = \mathrm{H}_1 \circ \mathrm{H}_2 \circ \mathrm{H}_3 \tag{5.5}$$

After applying the three healthiness conditions to S, the resulting expression is:

$$\begin{aligned} \mathrm{H}\left(S\right) = \{ \ & \left(A, O_1\right), \left(B, O_2\right), \\ & \left(A \wedge \neg B, O_3\right), \left(A \vee B, O_4\right), \\ & \left(\neg A \wedge \neg B, \mathrm{Nominal}\, undefined\right) \ \} \end{aligned}$$

The healthiness conditions are useful to faults modelling, aiding the faults analyst to check contradictions and completeness. Also, obtaining safe predicates is only possible in healthy expressions. In the next section, we show how to verify non-determinism in AL expressions.

5.3 Non-determinism

The non-determinism is usually an undesirable property. It causes an unexpected behaviour, so the analysis shall consider the activation of fault even if the fault might or not be active.

To identify ~~a~~ non-determinism, we can check for the negation of a contradiction in a pair of predicates in the inner algebra.

Definition 5.4 (Non-determinism)**.** *Let exp be an expression in AL.*

$$\begin{aligned} \text{nondeterministic}\,(exp) = & \exists\,(P_1, O_1)\,,(P_2, O_2) \in exp \bullet \\ & \neg\text{contradiction}\,(P_1 \wedge P_2) \end{aligned} \tag{5.6}$$

If there is at least one combination that evaluates $P_1 \wedge P_2$ to true (it is not a contradiction), then *exp* is non-deterministic. Our example is clearly non-deterministic as at least $A \wedge (A \vee B)$ is not a contradiction.

To analyse components and systems, and to model faults propagation, a predicates notation is shown in the next section. The predicates notation offers more two ways to check non-determinism.

5.4 Predicates Notation

The AL needs a special notation to enable the analysis of: (i) a particular faults expression, or (ii) a propagation in components. Such a special notation extracts predicates in the inner algebra given an output of interest.

Definition 5.5 (Predicate)**.** *Let exp be an expression in AL, and O_x an operational mode. A predicate over exp that matches O_x is then:*

$$\langle\!| \text{out}\,(exp) = O_x |\!\rangle \iff \exists\,(P, O) \in \text{H}\,(exp) \mid O = O_x \bullet P \tag{5.7}$$

The predicate notation function returns a predicate in the inner algebra. For the example in the beginning of this section, the predicate for O_2 is obtained as follows:

$$\langle\!| \text{out}\,(S) = O_2 |\!\rangle = B$$

To allow fault propagation of components we need another special notation that expands the modes of an expression with a predicate in the inner algebra.

Definition 5.6 (Modes)**.** *Let exp be an expression in AL, and P a predicate in the inner algebra, then:*

$$\text{modes}\,(exp, P) = \{(P_i \wedge P, O_i) \mid (P_i, O_i) \in \text{H}\,(exp)\} \tag{5.8}$$

Finally, to check the possible outputs, we need a function to obtain a set of outputs given an expression.

Definition 5.7 (Activation)**.** *Let exp be an expression in AL, and P_x a predicate in the inner algebra, then:*

$$\operatorname{activation}\left(exp, P_x\right) = \left\{O | (P, O) \in \mathrm{H}\left(exp\right) \wedge \operatorname{tautology}\left(P_x \implies P\right)\right\} \tag{5.9}$$

The non-determinism can also be checked using the predicates notation and the activation property:

$$\operatorname{activation}\left(S, A \wedge \neg B\right) = \left\{O_1, O_3\right\} \tag{5.10a}$$

$$\langle| \operatorname{out}\left(S\right) = O_1 |\rangle \wedge \langle| \operatorname{out}\left(S\right) = O_3 |\rangle = A \wedge \neg B \tag{5.10b}$$

Equation (5.10a) shows that both O_1 and O_3 can be observed if $A \wedge \neg B$ is *true*. Equation (5.10b) states that if the possible operational modes of healthy S are O_1 and O_3, then the predicate is $A \wedge \neg B$. In the next chapter, we show a practical case study using these properties and notations.

6 Case study

Our industrial partner provided us with the Simulink model of an Actuator Control System (depicted in Figure 25). The failure expression of this system (that is, for each of its constituent components) was also provided by our industrial partner (we show some of them in Table 7). In what follows we illustrate our strategy using the Monitor component.

A monitor component is a system commonly used for fault tolerance [89, 90]. Initially, the monitor connects the main input (power source on input port 1) with its output. It observes the value of this input port and compares it to a threshold. If the value is below the threshold, the monitor disconnects the output from the main input and connects to the secondary input. We present the Simulink model for this monitor in Figure 26.

Now we show four contributions: (i) using ATF, but only with Boolean operators, thus ignoring ordering, we can obtain the same results obtained in [28], (ii) representing each of the fault traces reported in [28] as a term in our proposed ATF, using the mapping function shown in Section 4.4, (iii) modelling faults of the monitor using AL, using expressions in Boolean Algebra, and (iv) modelling faults of the monitor with AL, but using ATF as the inner algebra. Similarly to the association of fault events of Table 7 in Section 3.5, we associate the fault events as:

$$
\begin{aligned}
b_1 &= \text{LowPower-In1} & \qquad B_1 &= \mathbf{var}\, b_1 \\
b_2 &= \text{LowPower-In2} & \qquad B_2 &= \mathbf{var}\, b_2 \\
f &= \text{SwitchFailure} & \qquad F &= \mathbf{var}\, f
\end{aligned}
$$

6.1 From traces to structure expressions with Boolean operators

In this section we show that the same result reported in [28] in terms of static failure expression (or Boolean propositions) can be obtained with our Boolean operator without using XBefore. Similarly to the mapping function shown in Section 4.4, we define a mapping function from traces to ATF with Boolean operators only as:

$$\langle[\ \ ]\rangle_{bool} = \top \tag{6.1a}$$

$$\langle[\ f\ ]\rangle_{bool} = \mathbf{var}\, f \tag{6.1b}$$

$$\langle[\ f_1\ ] @ tr\rangle_{bool} = \mathbf{var}\, f_1 \land \langle tr\rangle_{bool} \tag{6.1c}$$

$$\langle\{tr_1, tr_2, \ldots, tr_n\}\rangle_{bool} = \bigvee_{i\in\{1,\ldots,n\}} \left(\langle tr_i\rangle_{bool} \land \neg \bigvee_{j\in \mathrm{Gen}_{tr_i}} \mathbf{var}\, f_j\right) \tag{6.1d}$$

The only difference of the mapping function, when considering Boolean operators only, is Eq. (6.1c). Equations (6.1a), (6.1b) and (6.1d) are similar to Eqs. (4.37a), (4.37b) and (4.37d)

For each trace shown in Section 3.5, the mapping function generates the following expressions

$$
\begin{aligned}
\texttt{TRACE 1:}\quad & \langle[\ f, b_2\ ]\rangle_{bool} = F \wedge B_2 \\
\texttt{TRACE 2:}\quad & \langle[\ b_2, f\ ]\rangle_{bool} = B_2 \wedge F \\
\texttt{TRACE 3:}\quad & \langle[\ b_1, b_2\ ]\rangle_{bool} = B_1 \wedge B_2 \\
\texttt{TRACE 4:}\quad & \langle[\ b_2, b_1\ ]\rangle_{bool} = B_2 \wedge B_1 \\
\texttt{TRACE 5:}\quad & \langle[\ b_1, f\ ]\rangle_{bool} = B_1 \wedge F \\
\texttt{TRACE 6:}\quad & \langle[\ b_1, f, b_2\ ]\rangle_{bool} = B_1 \wedge F \wedge B_2 \\
\texttt{TRACE 7:}\quad & \langle[\ b_1, b_2, f\ ]\rangle_{bool} = B_1 \wedge B_2 \wedge F \\
\texttt{TRACE 8:}\quad & \langle[\ b_2, b_1, f\ ]\rangle_{bool} = B_2 \wedge B_1 \wedge F \\
\texttt{TRACE 9:}\quad & \langle[\ f, b_1, b_2\ ]\rangle_{bool} = F \wedge B_1 \wedge B_2 \\
\texttt{TRACE 10:}\quad & \langle[\ f, b_2, b_1\ ]\rangle_{bool} = F \wedge B_2 \wedge B_1 \\
\texttt{TRACE 11:}\quad & \langle[\ b_2, f, b_1\ ]\rangle_{bool} = B_2 \wedge F \wedge B_1
\end{aligned}
$$

They represent the same faults shown in Section 3.5. Applying mapping function, Eq. (6.1d), for the previously shown set of traces, we obtain following expression in ATF (and in FBA):

$$
M_{bool} = (B_1 \wedge B_2) \vee (F \wedge (B_1 \vee B_2)) \tag{6.2}
$$

which is equivalent to our industrial partner's failure expression shown in Table 7. This shows that ATF can represent (static) failure expression as in our previous work [28].

6.2 From traces to structure expressions with XBefore

Now, by using ATF with the XBefore operator and the mapping function shown in Eqs. (4.37a) to (4.37d), we can capture each possible individual sequences as generated by the work [28]:

$$\begin{aligned}
\texttt{TRACE 1:}\ & \langle[\ f, b_2\ ]\rangle_{XB} = (F \rightarrow B_2)\\
\texttt{TRACE 2:}\ & \langle[\ b_2, f\ ]\rangle_{XB} = (B_2 \rightarrow F)\\
\texttt{TRACE 3:}\ & \langle[\ b_1, b_2\ ]\rangle_{XB} = (B_1 \rightarrow B_2)\\
\texttt{TRACE 4:}\ & \langle[\ b_2, b_1\ ]\rangle_{XB} = (B_2 \rightarrow B_1)\\
\texttt{TRACE 5:}\ & \langle[\ b_1, f\ ]\rangle_{XB} = (B_1 \rightarrow F)\\
\texttt{TRACE 6:}\ & \langle[\ b_1, f, b_2\ ]\rangle_{XB} = B_1 \rightarrow (F \rightarrow B_2)\\
\texttt{TRACE 7:}\ & \langle[\ b_1, b_2, f\ ]\rangle_{XB} = B_1 \rightarrow (B_2 \rightarrow F)\\
\texttt{TRACE 8:}\ & \langle[\ b_2, b_1, f\ ]\rangle_{XB} = B_2 \rightarrow (B_1 \rightarrow F)\\
\texttt{TRACE 9:}\ & \langle[\ f, b_1, b_2\ ]\rangle_{XB} = F \rightarrow (B_1 \rightarrow B_2)\\
\texttt{TRACE 10:}\ & \langle[\ f, b_2, b_1\ ]\rangle_{XB} = F \rightarrow (B_2 \rightarrow B_1)\\
\texttt{TRACE 11:}\ & \langle[\ b_2, f, b_1\ ]\rangle_{XB} = B_2 \rightarrow (F \rightarrow B_1)
\end{aligned}$$

Using the mapping function, Eq. (4.37d), for the previously shown set of traces, we obtain:

$$\begin{aligned}
M_A = {} & (F \rightarrow B_2 \wedge \neg B_1) \vee (B_2 \rightarrow F \wedge \neg B_1) \vee (B_1 \rightarrow B_2 \wedge \neg F) \vee \\
& (B_2 \rightarrow B_1 \wedge \neg F) \vee (B_1 \rightarrow F \wedge \neg B_2) \vee (B_1 \rightarrow (F \rightarrow B_2)) \vee \\
& (B_1 \rightarrow (B_2 \rightarrow F)) \vee (B_2 \rightarrow (B_1 \rightarrow F)) \vee (F \rightarrow (B_1 \rightarrow B_2)) \vee \\
& (F \rightarrow (B_2 \rightarrow B_1)) \vee (B_2 \rightarrow (F \rightarrow B_1)) \\
= {} & (F \wedge B_2 \wedge \neg B_1) \vee (B_1 \wedge B_2 \wedge \neg F) \vee (B_1 \rightarrow F \wedge \neg B_2) \vee \\
& (B_1 \rightarrow (F \rightarrow B_2)) \vee (B_1 \rightarrow (B_2 \rightarrow F)) \vee (B_2 \rightarrow (B_1 \rightarrow F)) \vee \\
& (F \rightarrow (B_1 \rightarrow B_2)) \vee (F \rightarrow (B_2 \rightarrow B_1)) \vee (B_2 \rightarrow (F \rightarrow B_1)) \qquad \text{by Eq. (4.23c)} \\
= {} & (F \wedge B_2 \wedge \neg B_1) \vee (B_1 \wedge B_2 \wedge \neg F) \vee (B_1 \rightarrow F \wedge \neg B_2) \vee \\
& (B_2 \wedge (B_1 \rightarrow F)) \vee (B_2 \wedge (F \rightarrow B_1)) \qquad \text{by Eq. (4.24)} \\
= {} & (B_1 \wedge B_2) \vee (F \wedge B_2) \vee (B_1 \rightarrow F \wedge \neg B_2) \qquad (6.3)
\end{aligned}$$

The semantics of the above expression is: (i) fault b_2 ($\mathbf{var}\, b_2$) occurs and fault b_1 ($\mathbf{var}\, b_1$) or fault f ($\mathbf{var}\, f$) occurs, or (ii) fault b_1 occurs before fault f and fault b_2 does not occur, which is more precise than the expression found without considering order of events.

Expanding Eq. (6.2), we have:

$$(B_1 \wedge B_2) \vee (F \wedge B_2) \vee (F \wedge B_1)$$

which differ from Eq. (6.3) only on terms: $F \wedge B_1$ (of M_{bool}) and $F \rightarrow B_1 \wedge \neg B_2$ (of M_A).

6.3 From AL to structure expressions with Boolean operators

The power source has only two possible operational modes: (i) the power source works as expected, providing a nominal value of $12V$, and (ii) it has an internal failure B_i, and its operational mode is "low power". In AL it is modelled as:

$$PowerSource_i = \{(B_i, LP), (\neg B_i, \text{Nominal}\, 12V)\} \tag{6.4}$$

where LP is the LowPower failure. It is easy to see that the expression $PowerSource_i$ is healthy.

The monitor is a bit different because its behaviour depends not only on internal faults, but also on its inputs. We will now use the predicates notation defined in Section 5.4 to express fault propagation. As the monitor has two inputs and its behaviour is described in Figure 26, then it is a function of the expressions of both inputs:

$$\begin{aligned} &Monitor_{bool}(in_1, in_2) = \\ &\qquad \text{modes}(in_1, \langle|\text{out}(in_1) = \text{Nominal}\, X|\rangle \wedge \neg F) \cup \\ &\qquad \text{modes}(in_2, \neg\langle|\text{out}(in_1) = \text{Nominal}\, X|\rangle \wedge \neg F) \cup \\ &\qquad \text{modes}(in_2, \langle|\text{out}(in_1) = \text{Nominal}\, X|\rangle \wedge F) \cup \\ &\qquad \text{modes}(in_1, \neg\langle|\text{out}(in_1) = \text{Nominal}\, X|\rangle \wedge F) \end{aligned} \tag{6.5}$$

where X is an unbound variable and assumes any value. The expression states the following:

- The monitor output is the same as in_1 if the output of in_1 *is* nominal and *there is no* internal failure in the monitor:

$$\text{modes}(in_1, \langle|\text{out}(in_1) = \text{Nominal}\, X|\rangle \wedge \neg F)$$

- The monitor output is the same as in_2 if the output of in_1 *is not* nominal and *there is no* internal failure in the monitor:

$$\text{modes}(in_2, \neg\langle|\text{out}(in_1) = \text{Nominal}\, X|\rangle \wedge \neg F)$$

- The monitor output is the converse of the previous two conditions if the internal failure F is active:

$$\begin{aligned} \text{modes}(in_2, \langle|\text{out}(in_1) = \text{Nominal}\, X|\rangle \wedge F) \cup \\ \text{modes}(in_1, \neg\langle|\text{out}(in_1) = \text{Nominal}\, X|\rangle \wedge F) \end{aligned}$$

The operational modes (observed behaviour) of the monitor depend on: (i) its internal fault, and (ii) propagated errors from its inputs. Composing the monitor with the two power sources, we obtain the AL expression of a power supply subsystem $System_{bool}$:

$$
\begin{aligned}
& System_{bool} = \\
& \quad Monitor_{bool}\left(PowerSource_1, PowerSource_2\right) \\
& = \mathrm{modes}\left(in_1, \neg B_1 \wedge \neg F\right) \cup \mathrm{modes}\left(in_2, \neg\neg B_1 \wedge \neg F\right) \cup \\
& \quad \mathrm{modes}\left(in_2, \neg B_1 \wedge F\right) \cup \mathrm{modes}\left(in_1, \neg\neg B_1 \wedge F\right) && \text{by Eq. (5.7)} \\
& = \mathrm{modes}\left(in_1, \neg B_1 \wedge \neg F\right) \cup \mathrm{modes}\left(in_2, B_1 \wedge \neg F\right) \cup \\
& \quad \mathrm{modes}\left(in_2, \neg B_1 \wedge F\right) \cup \mathrm{modes}\left(in_1, B_1 \wedge F\right) && \text{by simplification} \\
& = \left\{\left(P_i \wedge \neg B_1 \wedge \neg F, O_i\right) \mid \left(P_i, O_i\right) \in in_1\right\} \cup \\
& \quad \left\{\left(P_i \wedge B_1 \wedge \neg F, O_i\right) \mid \left(P_i, O_i\right) \in in_2\right\} \cup \\
& \quad \left\{\left(P_i \wedge \neg B_1 \wedge F, O_i\right) \mid \left(P_i, O_i\right) \in in_2\right\} \cup \\
& \quad \left\{\left(P_i \wedge B_1 \wedge F, O_i\right) \mid \left(P_i, O_i\right) \in in_1\right\} && \text{by Eq. (5.8)} \\
& = \left\{\left(B_1 \wedge \neg B_1 \wedge \neg F, LP\right),\right. \\
& \quad \left(\neg B_1 \wedge \neg B_1 \wedge \neg F, \text{Nominal } 12V\right), \\
& \quad \left(B_2 \wedge B_1 \wedge \neg F, LP\right), \\
& \quad \left(\neg B_2 \wedge B_1 \wedge \neg F, \text{Nominal } 12V\right), \\
& \quad \left(B_2 \wedge \neg B_1 \wedge F, LP\right), \\
& \quad \left(\neg B_2 \wedge \neg B_1 \wedge F, \text{Nominal } 12V\right), \\
& \quad \left(B_1 \wedge B_1 \wedge F, LP\right), \\
& \quad \left.\left(\neg B_1 \wedge B_1 \wedge F, \text{Nominal } 12V\right)\right\} && \text{replacing vars}
\end{aligned}
$$

Simplifying and applying H_1, we obtain:

$$
\begin{aligned}
\mathrm{H}_1\left(System_{\mathrm{bool}}\right) = & \\
& \left\{\left(\neg B_1 \wedge \neg F, \text{Nominal } 12V\right), \left(B_2 \wedge B_1 \wedge \neg F, LP\right),\right. \\
& \left(\neg B_2 \wedge B_1 \wedge \neg F, \text{Nominal } 12V\right), \left(B_2 \wedge \neg B_1 \wedge F, LP\right), \\
& \left.\left(\neg B_2 \wedge \neg B_1 \wedge F, \text{Nominal } 12V\right), \left(B_1 \wedge F, LP\right)\right\}
\end{aligned}
$$

Applying, H_3, we simplify to:

$$
\begin{aligned}
&\mathrm{H}_3 \circ \mathrm{H}_1 \left(System_{\mathrm{bool}}\right) \\
&\quad = \left\{ \begin{pmatrix} \left(\neg B_1 \wedge \neg F\right) \vee \\ \left(B_1 \wedge \neg B_2 \wedge \neg F\right) \vee, \mathrm{Nominal}\, 12V \\ \left(\neg B_1 \wedge \neg B_2 \wedge F\right) \end{pmatrix}, \right. \\
&\qquad \left. \begin{pmatrix} \left(B_1 \wedge B_2 \wedge \neg F\right) \vee \\ \left(\neg B_1 \wedge B_2 \wedge F\right) \vee, LP \\ \left(B_1 \wedge F\right) \end{pmatrix} \right\} \\
&\quad = \left\{ \left(\left(\neg B_1 \wedge \neg B_2\right) \vee \neg F \wedge \left(\neg B_1 \vee \neg B_2\right), \mathrm{Nominal}\, 12V\right), \right. \\
&\qquad \left. \left(F \wedge \left(B_1 \vee B_2\right) \vee \left(B_1 \wedge B_2\right), LP\right) \right\}
\end{aligned}
$$

The monitor expression is H_2-healthy (the predicates are complete), thus:

$$\mathrm{H}_2 \circ \mathrm{H}_3 \circ \mathrm{H}_1 \left(System_{\mathrm{bool}}\right) = \mathrm{H}_3 \circ \mathrm{H}_1 \left(System_{\mathrm{bool}}\right)$$

The resulting expression for the monitor after applying all healthiness conditions is:

$$
\begin{aligned}
\mathrm{H} \left(System_{\mathrm{bool}}\right) = &\left\{ \left(\left(\neg B_1 \wedge \neg B_2\right) \vee \neg F \wedge \left(\neg B_1 \vee \neg B_2\right), \mathrm{Nominal}\, 12V\right), \right. \\
&\left. \left(F \wedge \left(B_1 \vee B_2\right) \vee \left(B_1 \wedge B_2\right), LP\right) \right\}
\end{aligned}
\tag{6.6}
$$

The operational modes of $System_{\mathrm{bool}}$ is either $\mathrm{Nominal}\, 12V$ or LP (low power).

Finally, we obtain the *low power* structure expression (see Table 7) using the predicates notation:

$$\langle\!| \mathrm{out} \left(System_{\mathrm{bool}}\right) = LP |\!\rangle \iff F \wedge \left(B_1 \vee B_2\right) \vee \left(B_1 \wedge B_2\right)$$

The monitor expression also indicates that if the switch is operational ($\neg F$) and at least one PowerSource is operational ($\neg B_1 \vee \neg B_2$), the monitor output is nominal. But if at least one PowerSource is faulty ($B_1 \vee B_2$) and the monitor has an internal failure (F) the system is not operational. These two sentences written in AL using predicates notation are:

$$
\begin{aligned}
&\mathrm{activation} \left(System_{\mathrm{bool}}, \neg F \wedge \left(\neg B_1 \vee \neg B_2\right)\right) \\
&\quad = \left\{ O \middle| \left(P, O\right) \in \mathrm{H} \left(System_{\mathrm{bool}}\right) \wedge \right. \\
&\qquad \left. \mathrm{tautology} \left(\neg F \wedge \left(\neg B_1 \vee \neg B_2\right) \implies P\right) \right\} && \text{[by Eq. (5.9)]} \\
&\quad = \left\{ \mathrm{Nominal}\, 12V \right\} && \text{[by simplification]} \qquad (6.7a) \\
&\mathrm{activation} \left(System_{\mathrm{bool}}, F \wedge \left(B_1 \vee B_2\right)\right) \\
&\quad = \left\{ O \middle| \left(P, O\right) \in \mathrm{H} \left(System_{\mathrm{bool}}\right) \wedge \right. \\
&\qquad \left. \mathrm{tautology} \left(F \wedge \left(B_1 \vee B_2\right) \implies P\right) \right\} && \text{[by Eq. (5.9)]} \\
&\quad = \left\{ LP \right\} && \text{[by simplification]} \qquad (6.7b)
\end{aligned}
$$

6.4 From AL to structure expressions with XBefore

Now, let's consider the same system but with a subtle modification. As shown in [52], the order of the occurrence of faults may be relevant, and the qualitative and quantitative analyses results may be different than those results without considering the order of the occurrence of faults. Observing Figure 26, we see that if F activates before a failure in the first input of the monitor, then it would display a nominal behaviour, because the internal failure F anticipates switching to the second input. On the other hand, if the first input fails before F, then the monitor would switch to the second input, then switch back, due to the internal failure. We obtain the following expression for the monitor, now using the ATF:

$$\begin{aligned} &Monitor_{XB}\left(in_1, in_2\right) = \\ &\qquad \text{modes}\left(in_1, \langle|\text{out}\left(in_1\right) = \text{Nominal}\, X|\rangle \wedge \neg F\right) \cup \\ &\qquad \text{modes}\left(in_2, \neg\langle|\text{out}\left(in_1\right) = \text{Nominal}\, X|\rangle \wedge \neg F\right) \cup \\ &\qquad \text{modes}\left(in_2, \langle|\text{out}\left(in_1\right) = \text{Nominal}\, X|\rangle \wedge F\right) \cup \\ &\qquad \text{modes}\left(in_1, \neg\langle|\text{out}\left(in_1\right) = \text{Nominal}\, X|\rangle \rightarrow F\right) \cup \\ &\qquad \text{modes}\left(in_2, F \rightarrow \neg\langle|\text{out}\left(in_1\right) = \text{Nominal}\, X|\rangle\right) \end{aligned} \tag{6.8}$$

where X is an unbound variable and assumes any value.

The difference to $System_{\text{bool}}$ (Eq. (6.5)) is only the finer analysis of the cases of erroneous behaviour of the first input and an internal failure. Note that the finer analysis splits the predicate

$$\neg\langle|\text{out}\left(in_1\right) = \text{Nominal}\, 12V|\rangle \wedge F \qquad \text{(activates } in_1\text{)}$$

into:

$$\neg\langle|\text{out}\left(in_1\right) = \text{Nominal}\, 12V|\rangle \rightarrow F \qquad \text{(activates } in_1\text{)}$$

and

$$F \rightarrow \neg\langle|\text{out}\left(in_1\right) = \text{Nominal}\, 12V|\rangle \qquad \text{(activates } in_2\text{)}$$

We can assure that such a split is complete because the predicate notation evaluates to B_1. Thus the operands satisfy all temporal properties (Eqs. (4.14a) to (4.14d)) and events independence (Eq. (4.19)), ~~thus~~ the law shown in Eq. (4.21b) is valid. For case (i), the expected behaviour is the same as in_1 because the system switches to in_2, but then an internal failure occurs, and it switches back to in_1. For case (ii), it switches to in_2 due to an internal failure, then the first input fails, so the behaviour is similar to the nominal behaviour (see the second *modes* in Eq. (6.8)).

Following the similar expansions of Eq. (6.5), we obtain:

$$
\begin{aligned}
System_{XB} = & Monitor_{XB}\left(PowerSource_1, PowerSource_2\right)\\
= & \{\left(B_1 \wedge \neg B_1 \wedge \neg F, LP\right),\\
& \left(\neg B_1 \wedge \neg B_1 \wedge \neg F, \text{Nominal}\,12V\right),\\
& \left(B_2 \wedge B_1 \wedge \neg F, LP\right),\\
& \left(\neg B_2 \wedge B_1 \wedge \neg F, \text{Nominal}\,12V\right),\\
& \left(B_2 \wedge \neg B_1 \wedge F, LP\right),\\
& \left(\neg B_2 \wedge \neg B_1 \wedge F, \text{Nominal}\,12V\right),\\
& \left(B_1 \wedge B_1 \rightarrow F, LP\right),\\
& \left(\neg B_1 \wedge B_1 \rightarrow F, \text{Nominal}\,12V\right)\},\\
& \left(B_2 \wedge F \rightarrow B_1, LP\right),\\
& \left(\neg B_2 \wedge F \rightarrow B_1, \text{Nominal}\,12V\right)\}
\end{aligned}
$$

Simplifying and applying H_1 to remove contradictions, we obtain:

$$
\begin{aligned}
& \mathrm{H}_1\left(System_{XB}\right) =\\
& \qquad \{\left(\neg B_1 \wedge \neg F, \text{Nominal}\,12V\right), \left(B_2 \wedge B_1 \wedge \neg F, LP\right),\\
& \qquad \left(\neg B_2 \wedge B_1 \wedge \neg F, \text{Nominal}\,12V\right), \left(B_2 \wedge \neg B_1 \wedge F, LP\right),\\
& \qquad \left(\neg B_2 \wedge \neg B_1 \wedge F, \text{Nominal}\,12V\right), \left(B_1 \rightarrow F, LP\right),\\
& \qquad \left(B_2 \wedge F \rightarrow B_1, LP\right), \left(\neg B_2 \wedge F \rightarrow B_1, \text{Nominal}\,12V\right)\}
\end{aligned}
$$

Applying H_3 to remove redundant terms with identical operational modes and using the rules shown in Section 4.2, we simplify to:

$$
\begin{aligned}
&\mathrm{H}_3 \circ \mathrm{H}_1 \left(System_{XB}\right) \\
&= \left\{ \left(\begin{array}{c} (\neg B_1 \wedge \neg F) \vee \\ (B_1 \wedge \neg B_2 \wedge \neg F) \vee \\ (\neg B_1 \wedge \neg B_2 \wedge F) \vee \\ (\neg B_2 \wedge F \rightarrow B_1) \end{array}, \mathrm{Nominal}\, 12V \right), \right. \\
&\qquad \left. \left(\begin{array}{c} (B_1 \wedge B_2 \wedge \neg F) \vee \\ (\neg B_1 \wedge B_2 \wedge F) \vee \\ (B_1 \rightarrow F) \vee \\ (B_2 \wedge F \rightarrow B_1) \end{array}, LP \right) \right\} \\
&= \left\{ \left(\begin{array}{c} (\neg B_1 \wedge \neg F) \vee \\ (B_1 \wedge \neg B_2 \wedge \neg F) \vee \\ (\neg B_1 \wedge \neg B_2 \wedge F) \vee \\ (\neg B_2 \wedge F \rightarrow B_1) \end{array}, \mathrm{Nominal}\, 12V \right), \right. \\
&\qquad \left. \left(\begin{array}{c} (B_1 \wedge B_2 \wedge \neg F) \vee \\ (\neg B_1 \wedge B_2 \wedge F) \vee \\ (B_2 \wedge B_1 \rightarrow F) \vee, LP \\ (\neg B_2 \wedge B_1 \rightarrow F) \vee \\ (B_2 \wedge F \rightarrow B_1) \end{array} \right) \right\} \\
&= \{ ((\neg B_1 \wedge \neg B_2) \vee \neg F \wedge (\neg B_1 \vee \neg B_2) \vee \\
&\quad \neg B_2 \wedge F \rightarrow B_1, \mathrm{Nominal}\, 12V), \\
&\quad ((B_1 \wedge B_2) \vee (B_2 \wedge F) \vee (\neg B_2 \wedge B_1 \rightarrow F), LP)\}
\end{aligned}
$$

The monitor expression is H_2-healthy. Simplifying Boolean operators as usual, the XBefore expression is:

$$(\neg B_2 \wedge F \rightarrow B_1) \vee (\neg B_2 \wedge B_1 \rightarrow F)$$

which simplifies to

$$\neg B_2 \wedge \ F \wedge B_1 \qquad \text{by Eq. (4.21b)}$$

Thus:

$$\mathrm{H}_2 \circ \mathrm{H}_3 \circ \mathrm{H}_1 \left(System_{XB}\right) = \mathrm{H}_3 \circ \mathrm{H}_1 \left(System_{XB}\right)$$

The resulting expression for the monitor after applying all healthiness conditions is:

$$
\begin{aligned}
\mathrm{H}\left(System_{XB}\right) = \{ & ((\neg B_1 \wedge \neg B_2) \vee \neg F \wedge (\neg B_1 \vee \neg B_2) \vee \\
& \neg B_2 \wedge F \rightarrow B_1, \text{Nominal}\, 12V), \\
& ((B_1 \wedge B_2) \vee (B_2 \wedge F) \vee \\
& (\neg B_2 \wedge B_1 \rightarrow F), LP)\}
\end{aligned}
\tag{6.9}
$$

Finally, we obtain the *low power* structure expression of the monitor using the predicates notation:

$$\langle | \mathrm{out}\left(System_{XB}\right) = LP | \rangle \iff (B_1 \wedge B_2) \vee (B_2 \wedge F) \vee (\neg B_2 \wedge B_1 \rightarrow F)$$

Thus, $System_{XB}$ fails with LP if:

- Both power sources fail;
- The monitor fails to detect the nominal state of the first power source and the second power source is in a failure state;
- The monitor fails to detect the failure state of the first power source (the monitor fails after the failure of the first power source).

Note that if the monitor fails before the failure of the first power source, it fails to detect the operational mode of the first power source and switches to the second power source, which is in a nominal state (see expression $\neg B_2 \wedge F \rightarrow B_1$ in Eq. (6.9)).

7 Conclusion

In this work we presented a foundational theory to support a more precise representation of fault events as compared to our previous strategy for injecting faults [28]. The failure expression is essential for system safety assessment because it is used as basic input for building fault trees [25, 31, 91]. Furthermore, we still connect the strategy presented in [92] with the works reported in [31] (functional analysis) and in [91, 25] (safety assessment) because our new algebra is at least a Boolean algebra.

The work reported in [21, 20, 33] tackles simultaneity with “nearly simultaneous” events [93]. But we consider instantaneous events, like the work reported in [23], because we assume that simultaneity is probabilistically impossible.

7.1 Future work

Boolean formulas reduction can be achieved by: (i) application of Boolean laws, (ii) BDD, or (iii) FBAs. We used Boolean and XBefore laws to reduce ATF formulas. The work reported in [42, 43] uses Sequential BDDs to reduce formulas with order-based operators. We plan to use similar concepts in a future work.

The work reported in [7] states that DTMCs (Markov chain) is more appropriate to represent several states than SFTs. Considering that DFTs were conceived as a visual representation of Markov chains, then we may say that DFTs can be used to represent several states. Thus they are suitable to propose the architectural model modifications as shown in Figure 4. The definition and the theory of “Fault Modelling and Fault Tolerance Patterns” and the automatic proposal of “Architectural Model Modifications” blocks are left as future work.

Bibliography

1 DUGAN, J. B.; BAVUSO, S. J.; BOYD, M. A. Dynamic fault-tree models for fault-tolerant computer systems. *Reliability, IEEE Transactions on*, v. 41, n. 3, p. 363 –377, sep 1992. ISSN 0018-9529.

2 BOYD, M. A. *Dynamic Fault Tree Models: Techniques for Analysis of Advanced Fault Tolerant Computer Systems.* Tese (Doutorado) — Duke University, Durham, NC, USA, 1992. UMI Order No. GAX92-02503.

3 MERLE, G. *Algebraic modelling of Dynamic Fault Trees, contribution to qualitative and quantitative analysis.* Tese (Theses) — École normale supérieure de Cachan - ENS Cachan, jul. 2010. Available from Internet: <https://tel.archives-ouvertes.fr/tel-00502012>.

4 ANAC. *Aeronautical Product Certification (in portuguese).* 2011. DOU Nº 230, Seção 1, p. 28, 01/12/2011. Available from Internet: <http://www2.anac.gov.br/biblioteca/resolucao/2011/RBAC21EMD01.pdf>.

5 FAA. Book, Online. *RTCA, Inc., Document RTCA/DO-178B.* [S.l.]: U.S. Dept. of Transportation, Federal Aviation Administration, [Washington, D.C.] :, 1993. [1] p. : p.

6 FAA. *Part 25 - Airworthiness Standards: Transport Category Airplanes.* [S.l.], 2007.

7 SAE. Miscellaneous, *SAE ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment.* [S.l.]: Society of Automotive Engineers (SAE), 1996.

8 AVIZIENIS, A.; LAPRIE, J.-C.; RANDELL, B.; LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, v. 1, n. 1, p. 11–33, 2004. ISSN 1545-5971.

9 ANDREWS, J. D. The use of not logic in fault tree analysis. *Quality and Reliability Engineering International*, John Wiley & Sons, Ltd., v. 17, n. 3, p. 143–150, 2001. ISSN 1099-1638. Available from Internet: <http://dx.doi.org/10.1002/qre.405>.

10 ANDREWS, J.; BEESON, S. Birnbaum's measure of component importance for noncoherent systems. *IEEE Transactions on Reliability*, Institute of Electrical & Electronics Engineers (IEEE), v. 52, n. 2, p. 213–219, jun 2003. Available from Internet: <http://dx.doi.org/10.1109/TR.2003.809656>.

11 OLIVA, S. Non-Coherent Fault Trees Can Be Misleading. *e-Journal of System Safety*, v. 42, n. 3, May-June 2006. Accessed in 13/jan/2016. Available from Internet: <http://www.system-safety.org/ejss/past/mayjune2006ejss/spotlight2_p1.php>.

12 CONTINI, S.; COJAZZI, G.; RENDA, G. On the use of non-coherent fault trees in safety and security studies. *Reliability Engineering & System Safety*, v. 93, n. 12, p. 1886 – 1895, 2008. ISSN 0951-8320. 17th European Safety and Reliability Conference. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0951832008001117>.

13 VAURIO, J. K. Importances of components and events in non-coherent systems and risk models. *Reliability Engineering & System Safety*, v. 147, p. 117 – 122, 2016. ISSN 0951-8320. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0951832015003348>.

14 AKERS. Binary Decision Diagrams. *IEEE Transactions on Computers*, Institute of Electrical & Electronics Engineers (IEEE), C-27, n. 6, p. 509–516, jun 1978.

15 BOUTE, R. The binary decision machine as programmable controller. *Euromicro Newsletter*, Elsevier BV, v. 2, n. 1, p. 16–22, jan 1976.

16 GIVANT, S.; HALMOS, P. *Introduction to Boolean Algebras*. [s.n.], 2009. XIV. (Undergraduate Texts in Mathematics, XIV). ISBN 978-0-387-68436-9. Available from Internet: <http://www.springer.com/mathematics/book/978-0-387-40293-2>.

17 ERICSON II, C. A. *Hazard Analysis Techniques for System Safety*. Wiley-Interscience, 2005. ISBN 978-0-471-72019-5. Available from Internet: <http://www.amazon.com/Hazard-Analysis-Techniques-System-Safety/dp/0471720194%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0471720194>.

18 VESELY, W.; GOLDBERG, F.; ROBERTS, N.; HAASL, D. *Fault Tree Handbook*. US Independent Agencies and Commissions, 1981. ISBN 9780160055829. Available from Internet: <http://www.nrc.gov/reading-rm/doc-collections/nuregs/staff/sr0492/>.

19 WALKER, M.; PAPADOPOULOS, Y. Synthesis and analysis of temporal fault trees with PANDORA: The time of Priority AND gates. *Nonlinear Analysis: Hybrid Systems*, v. 2, n. 2, p. 368 – 382, 2008. ISSN 1751-570X. Proceedings of the International Conference on Hybrid Systems and Applications, Lafayette, LA, USA, May 2006: Part II.

20 WALKER, M.; PAPADOPOULOS, Y. Qualitative temporal analysis: Towards a full implementation of the Fault Tree Handbook. *Control Engineering Practice*, v. 17, n. 10, p. 1115 – 1125, 2009. ISSN 0967-0661.

21 WALKER, M. D. *Pandora: a logic for the qualitative analysis of temporal fault trees*. Tese (Doutorado) — University of Hull, May 2009. Available from Internet: <https://hydra.hull.ac.uk/resources/hull:2526>.

22 MERLE, G.; ROUSSEL, J.-M.; LESAGE, J.-J. Algebraic determination of the structure function of Dynamic Fault Trees. *Reliability Engineering & System Safety*, Elsevier BV, v. 96, n. 2, p. 267–277, Feb 2011. ISSN 0951-8320.

23 MERLE, G.; ROUSSEL, J.-M.; LESAGE, J.-J. Quantitative Analysis of Dynamic Fault Trees Based on the Structure Function. *Quality and Reliability Engineering International*, Wiley-Blackwell, v. 30, n. 1, p. 143–156, Feb 2014. ISSN 0748-8017.

24 MERLE, G.; ROUSSEL, J.-M.; LESAGE, J.-J. Dynamic fault tree analysis based on the structure function. *2011 Proceedings - Annual Reliability and Maintainability Symposium*, IEEE, Jan 2011. Available from Internet: <http://dx.doi.org/10.1109/RAMS.2011.5754452>.

25 PAPADOPOULOS, Y.; MCDERMID, J.; SASSE, R.; HEINER, G. Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. *Reliability Engineering & System Safety*, v. 71, n. 3, p. 229–247, 2001. ISSN 0951-8320.

26 FEILER, P. H.; GLUCH, D. P.; HUDAK, J. J. The Architecture Analysis & Design Language (AADL): An Introduction. n. February, p. CMU/SEI–2006–TN–011, 2006. Available from Internet: <http://www.sei.cmu.edu/library/abstracts/reports/06tn011.cfm>.

27 DIDIER, A. *Estratégia sistemática para identificar falhas em componentes de hardware usando comportamento nominal.* Dissertação (Mestrado) — Universidade Federal de Pernambuco, 2 2012.

28 DIDIER, A.; MOTA, A. Identifying Hardware Failures Systematically. In: GHEYI, R.; NAUMANN, D. (Ed.). *Formal Methods: Foundations and Applications.* [S.l.]: Springer Berlin / Heidelberg, 2012, (Lecture Notes in Computer Science, v. 7498). p. 115–130. ISBN 978-3-642-33295-1.

29 SNOOKE, N.; PRICE, C. Model-driven automated software FMEA. In: *Reliability and Maintainability Symposium.* [S.l.: s.n.], 2011. p. 1–6. ISSN 0149-144X.

30 NISE, N. S. *Control systems engineering.* Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1992. ISBN 0-8053-5420-4.

31 JESUS, J.; MOTA, A.; SAMPAIO, A.; GRIJO, L. Architectural Verification of Control Systems Using CSP. In: QIN, S.; QIU, Z. (Ed.). *ICFEM.* [S.l.]: Springer, 2011. (Lecture Notes in Computer Science, v. 6991), p. 323–339. ISBN 978-3-642-24558-9.

32 MANIAN, R.; COPPIT, D.; SULLIVAN, K.; DUGAN, J. B. Bridging the gap between systems and dynamic fault tree models. In: *Reliability and Maintainability Symposium, 1999. Proceedings. Annual.* [S.l.: s.n.], 1999. p. 105 –111.

33 WALKER, M.; PAPADOPOULOS, Y. A hierarchical method for the reduction of temporal expressions in Pandora. In: *Proceedings of the First Workshop on DYnamic Aspects in DEpendability Models for Fault-Tolerant Systems.* New York, NY, USA: ACM, 2010. (DYADEM-FTS '10), p. 7–12. ISBN 978-1-60558-916-9.

34 LIU, L.; HASAN, O.; TAHAR, S. Formal Reasoning About Finite-State Discrete-Time Markov Chains in HOL. *J. Comput. Sci. Technol.*, Springer Science + Business Media, v. 28, n. 2, p. 217–231, mar 2013. Available from Internet: <http://dx.doi.org/10.1007/s11390-013-1324-6>.

35 COPPIT, D.; SULLIVAN, K. J.; DUGAN, J. B. Formal semantics of models for computational engineering: a case study on dynamic fault trees. In: *Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on.* [S.l.: s.n.], 2000. p. 270 –282. ISSN 1071-9458.

36 BOBBIO, A.; RAITERI, D. C.; MONTANI, S.; PORTINALE, L.; VARESIO, M. *DBNet, a tool to convert Dynamic Fault Trees to Dynamic Bayesian Networks.* [S.l.], 2005.

37 SERICOLA, B. Discrete-Time Markov Chains. In: *Markov Chains.* Wiley-Blackwell, 2013. p. 1–87. Available from Internet: <http://dx.doi.org/10.1002/9781118731543.ch1>.

38 IANNELLI, M.; PUGLIESE, A. An Introduction to Mathematical Population Dynamics: Along the trail of Volterra and Lotka. In: _____. Cham: Springer International Publishing, 2014. cap. Continuous-time Markov chains, p. 329–334. ISBN 978-3-319-03026-5. Available from Internet: <http://dx.doi.org/10.1007/978-3-319-03026-5_13>.

39 ANDERSON, W. J. *Continuous-Time Markov Chains*. Springer New York, 2012. Available from Internet: <http://www.ebook.de/de/product/25435927/william_j_anderson_continuous_time_markov_chains.html>.

40 BUCHHOLZ, P.; KATOEN, J.-P.; KEMPER, P.; TEPPER, C. Model-checking large structured Markov chains. *The Journal of Logic and Algebraic Programming*, Elsevier BV, v. 56, n. 1-2, p. 69–97, may 2003. Available from Internet: <http://dx.doi.org/10.1016/S1567-8326(02)00067-X>.

41 BAIER, C.; HAVERKORT, B.; HERMANNS, H.; KATOEN, J.-P. Model-checking algorithms for continuous-time markov chains. *IEEE Transactions on Software Engineering*, Institute of Electrical & Electronics Engineers (IEEE), v. 29, n. 6, p. 524–541, jun 2003. Available from Internet: <http://dx.doi.org/10.1109/TSE.2003.1205180>.

42 TANNOUS, O.; XING, L.; DUGAN, J. B. Reliability analysis of warm standby systems using sequential BDD. *2011 Proceedings - Annual Reliability and Maintainability Symposium*, IEEE, Jan 2011.

43 XING, L.; TANNOUS, O.; DUGAN, J. B. Reliability Analysis of Nonrepairable Cold-Standby Systems Using Sequential Binary Decision Diagrams. *IEEE Trans. Syst., Man, Cybern. A*, Institute of Electrical & Electronics Engineers (IEEE), v. 42, n. 3, p. 715–726, May 2012. ISSN 1558-2426.

44 MURPHY, K. P. *Dynamic bayesian networks: representation, inference and learning.* Tese (Doutorado) — University of California, Berkeley, 2002.

45 BRYANT. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, Institute of Electrical & Electronics Engineers (IEEE), C-35, n. 8, p. 677–691, aug 1986. Available from Internet: <http://dx.doi.org/10.1109/TC.1986.1676819>.

46 MIKULAK, R.; MCDERMOTT, R.; BEAUREGARD, M. *The Basics of FMEA, 2nd Edition*. CRC Press, 2008. ISBN 9781439809617. Available from Internet: <https://books.google.com.br/books?id=rM5Vi_0K9bUC>.

47 NIPKOW, T.; PAULSON, L. C.; WENZEL, M. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002. v. 2283. (LNCS, v. 2283). Available from Internet: <https://isabelle.in.tum.de/>.

48 ANDREWS, Z.; PAYNE, R.; ROMANOVSKY, A.; DIDIER, A.; MOTA, A. Model-based development of fault tolerant systems of systems. In: *Systems Conference (SysCon), 2013 IEEE International*. [S.l.: s.n.], 2013. p. 356–363.

49 ANDREWS, Z.; DIDIER, A.; PAYNE, R.; INGRAM, C.; HOLT, J.; PERRY, S.; OLIVEIRA, M.; WOODCOCK, J.; MOTA, A.; ROMANOVSKY, A. *Report on Timed Fault Tree Analysis — Fault modelling*. [S.l.], 2013. Available from Internet: <http://www.compass-research.eu/Project/Deliverables/D242.pdf>.

50 Object Management Group (OMG). *Systems Modelling Language (SysML) 1.3*. 2012. Website. Available from Internet: <http://www.omg.org/spec/SysML/1.3>.

51 MAIER, M. W. Architecting principles for systems-of-systems. *Systems Engineering*, John Wiley & Sons, Inc., v. 1, n. 4, p. 267–284, 1998. ISSN 1520-6858.

52 DIDIER, A.; MOTA, A. An Algebra of Temporal Faults. *Information Systems Frontiers*, jan 2016. ISSN 1572-9419.

53 JASKELIOFF, M.; MERZ, S. Proving the Correctness of Disk Paxos. *Archive of Formal Proofs*, jun. 2005. ISSN 2150-914x. <http://afp.sf.net/entries/DiskPaxos.shtml>, Formal proof development.

54 SOMMERVILLE, I. *Software Engineering*. Pearson, 2011. (International Computer Science Series). ISBN 9780137053469. Available from Internet: <http://books.google.com.br/books?id=l0egcQAACAAJ>.

55 CARVALHO, G.; BARROS, F.; CARVALHO, A.; CAVALCANTI, A.; MOTA, A.; SAMPAIO, A. NAT2TEST Tool: From Natural Language Requirements to Test Cases Based on CSP. In: *Software Engineering and Formal Methods*. Springer Science + Business Media, 2015. p. 283–290. Available from Internet: <http://dx.doi.org/10.1007/978-3-319-22969-0_20>.

56 AVRESKY, D.; ARLAT, J.; LAPRIE, J.-C.; CROUZET, Y. Fault injection for formal testing of fault tolerance. *IEEE Transactions on Reliability*, Institute of Electrical & Electronics Engineers (IEEE), v. 45, n. 3, p. 443–455, 1996. Available from Internet: <http://dx.doi.org/10.1109/24.537015>.

57 BRYANS, J.; CANHAM, S.; WOODCOCK, J. *CML Definition 4*. [S.l.], 2014. Available from Internet: <http://www.compass-research.eu/Project/Deliverables/D23.5-final-version.pdf>.

58 ROSCOE, A. W. *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997. Paperback. ISBN 0136744095.

59 MODARRES, M.; KAMINSKIY, M. P.; KRIVTSOV, V. *Reliability engineering and risk analysis: a practical guide*. [S.l.]: CRC press, 2009. ISBN 1420047051, 9781420047059.

60 DISTEFANO, S.; PULIAFITO, A. Dependability Evaluation with Dynamic Reliability Block Diagrams and Dynamic Fault Trees. *IEEE Transactions on Dependable and Secure Computing*, Institute of Electrical & Electronics Engineers (IEEE), v. 6, n. 1, p. 4–17, jan 2009. Available from Internet: <http://dx.doi.org/10.1109/TDSC.2007.70242>.

61 STAMATELATOS, M.; VESELY, W.; DUGAN, J.; FRAGOLA, J.; MINARICK III, J.; RAILSBACK, J. *Fault Tree Handbook with Aerospace Applications*. Washington, DC 20546, 2002. Available from Internet: <http://www.hq.nasa.gov/office/codeq/doctree/fthb.pdf>.

62 ADACHI, M.; PAPADOPOULOS, Y.; SHARVIA, S.; PARKER, D.; TOHDO, T. An approach to optimization of fault tolerant architectures using HiP-HOPS. *Software: Practice and Experience*, John Wiley & Sons, Ltd., v. 41, n. 11, p. 1303–1327, 2011. ISSN 1097-024X.

63 PALSHIKAR, G. K. Temporal fault trees. *Information and Software Technology*, v. 44, n. 3, p. 137 – 150, 2002. ISSN 0950-5849.

64 TANG, Z.; DUGAN, J. Minimal cut set/sequence generation for dynamic fault trees. In: *Reliability and Maintainability, 2004 Annual Symposium - RAMS*. [S.l.: s.n.], 2004. p. 207–213.

65 MERLE, G.; ROUSSEL, J.-M.; LESAGE, J.-J.; BOBBIO, A. Probabilistic Algebraic Analysis of Fault Trees With Priority Dynamic Gates and Repeated Events. *IEEE Trans. Rel.*, Institute of Electrical & Electronics Engineers (IEEE), v. 59, n. 1, p. 250–261, Mar 2010. ISSN 1558-1721.

66 PEARL, J. *Bayesian Networks: a model of self-activated memory for evidential reasoning*. [S.l.], 1985. Available from Internet: <ftp://ftp.cs.ucla.edu/pub/stat_ser/r43-1985.pdf>.

67 CHIOLA, G.; DUTHEILLET, C.; FRANCESCHINIS, G.; HADDAD, S. Stochastic well-formed colored nets and symmetric modeling applications. *IEEE Transactions on Computers*, Institute of Electrical & Electronics Engineers (IEEE), v. 42, n. 11, p. 1343–1360, 1993. Available from Internet: <http://dx.doi.org/10.1109/12.247838>.

68 JENSEN, K. Coloured Petri Nets. In: *Petri Nets: Central Models and Their Properties*. Springer Science + Business Media, 1987. p. 248–299. Available from Internet: <http://dx.doi.org/10.1007/978-3-540-47919-2_10>.

69 BOBBIO, A.; RAITERI, D. Parametric fault trees with dynamic gates and repair boxes. In: *Reliability and Maintainability, 2004 Annual Symposium - RAMS*. [S.l.: s.n.], 2004. p. 459–465.

70 SCHELLHORN, G.; THUMS, A.; REIF, W. *Formal Fault Tree Semantics*. 2002.

71 MOSZKOWSKI, B. *A Temporal Logic for Multi-Level Reasoning About Hardware,*. [S.l.], 1982. Available from Internet: <http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA324174>.

72 MAHMUD, N.; PAPADOPOULOS, Y.; WALKER, M. A translation of State Machines to temporal fault trees. *2010 International Conference on Dependable Systems and Networks Workshops (DSN-W)*, IEEE, Jun 2010. Available from Internet: <http://dx.doi.org/10.1109/DSNW.2010.5542620>.

73 MAHMUD, N.; WALKER, M.; PAPADOPOULOS, Y. Compositional Synthesis of Temporal Fault Trees from State Machines. *SIGMETRICS Perform. Eval. Rev.*, ACM, New York, NY, USA, v. 39, n. 4, p. 79–88, abr. 2012. ISSN 0163-5999.

74 SPIVEY, J. M. *The Z Notation: A Reference Manual*. Second edition. Prentice Hall International (UK) Ltd, 1998. Available from Internet: <http://spivey.oriel.ox.ac.uk/~mike/zrm/>.

75 GULATI, R.; DUGAN, J. A modular approach for analyzing static and dynamic fault trees. In: *Reliability and Maintainability Symposium. 1997 Proceedings, Annual*. [S.l.: s.n.], 1997. p. 57 –63.

76 SIMEU-ABAZI, Z.; LEFEBVRE, A.; DERAIN, J.-P. A methodology of alarm filtering using dynamic fault tree. *Reliability Engineering & System Safety*, Elsevier BV, v. 96, n. 2, p. 257–266, Feb 2011. ISSN 0951-8320.

77 JENSEN, K. High-Level Petri Nets. In: *Applications and Theory of Petri Nets*. Springer Science + Business Media, 1983. p. 166–180. Available from Internet: <http://dx.doi.org/10.1007/978-3-642-69028-0_12>.

78 BRACE, K. S.; RUDELL, R. L.; BRYANT, R. E. Efficient implementation of a BDD package. In: *Conference proceedings on 27th ACM/IEEE design automation conference - DAC '90*. Association for Computing Machinery (ACM), 1990. Available from Internet: <http://dx.doi.org/10.1145/123186.123222>.

79 RUDELL, R. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In: *Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1993. (ICCAD '93), p. 42–47. ISBN 0-8186-4490-7. Available from Internet: <http://dl.acm.org/citation.cfm?id=259794.259802>.

80 KISSMANN, P.; HOFFMANN, J. BDD Ordering Heuristics for Classical Planning. *Journal of Artificial Intelligence Research*, v. 51, 2014. Available from Internet: <http://doi.org/10.1613/jair.4586>.

81 STOLL, R. R. *Set Theory and Logic*. Dover Publications, 1979. (Dover books on advanced mathematics). ISBN 9780486638294. Available from Internet: <https://books.google.com.br/books?id=3-nrPB7BQKMC>.

82 MATHWORKS. *Simulink®*. 2010. Available from Internet: <http://www.mathworks.com/products/simulink>.

83 MATHWORKS. *Matlab®*. 2010. Available from Internet: <http://www.mathworks.com/products/matlab>.

84 ASTROM, K. J.; MURRAY, R. M. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton, NJ, USA: Princeton University Press, 2008. ISBN 0691135762, 9780691135762.

85 HUFFMAN, B. Free Boolean Algebra. *Archive of Formal Proofs*, v. 2010, mar. 2010. ISSN 2150-914x. Available from Internet: <http://afp.sourceforge.net/entries/Free-Boolean-Algebra.shtml>.

86 DIDIER, A. L. R.; MOTA, A. A Lattice-Based Representation of Temporal Failures. In: *Information Reuse and Integration (IRI), 2015 IEEE International Conference on*. [S.l.: s.n.], 2015. p. 295–302.

87 FUSSELL, J.; ABER, E.; RAHL, R. On the Quantitative Analysis of Priority-AND Failure Logic. *IEEE Transactions on Reliability*, R-25, n. 5, p. 324 – 326, 1976.

88 HOARE, C. A. R.; HE, J. *Unifying Theories of Programming*. Prentice Hall Englewood Cliffs, 1998. v. 14. Available from Internet: <http://www.unifyingtheories.org/>.

89 O'CONNOR, P.; NEWTON, D.; BROMLEY, R. *Practical reliability engineering*. [S.l.]: Wiley, 2002. ISBN 9780470844632.

90 KOREN, I.; KRISHNA, C. M. *Fault Tolerant Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN 0120885255.

91 GOMES, A.; MOTA, A.; SAMPAIO, A.; FERRI, F.; BUZZI, J. Systematic Model-Based Safety Assessment Via Probabilistic Model Checking. In: MARGARIA, T.; STEFFEN, B. (Ed.). *ISoLA (1)*. [S.l.]: Springer, 2010. (Lecture Notes in Computer Science, v. 6415), p. 625–639. ISBN 978-3-642-16557-3.

92 MOTA, A.; JESUS, J.; GOMES, A.; FERRI, F.; WATANABE, E. Evolving a Safe System Design Iteratively. In: SCHOITSCH, E. (Ed.). *SAFECOMP*. [S.l.]: Springer, 2010. (Lecture Notes in Computer Science, v. 6351), p. 361–374. ISBN 978-3-642-15650-2.

93 EDIFOR, E.; WALKER, M.; GORDON, N. Quantification of Simultaneous-AND Gates in Temporal Fault Trees. In: ZAMOJSKI, W.; MAZURKIEWICZ, J.; SUGIER, J.; WALKOWIAK, T.; KACPRZYK, J. (Ed.). *New Results in Dependability and Computer Systems*. [S.l.]: Springer International Publishing, 2013, (Advances in Intelligent Systems and Computing, v. 224). p. 141–151. ISBN 978-3-319-00944-5.

94 HAFTMANN, F.; LOCHBIHLER, A. *Dlist theory*. Available from Internet: <http://isabelle.in.tum.de/library/HOL/HOL-Quickcheck_Examples/Dlist.html>.

Appendix

APPENDIX A – Formal proofs in Isabelle/HOL

In the following we list all theorems and proofs concerning the laws presented in Chapter 4. The complete set of verifiable theory files is available at <http://www.cin.ufpe.br/~alrd/phd/phd-alrd.zip> (password: 6Zvq$5Vyj). We list only those files created in our work. Each theorem, proof or corollary is followed by its own proof.

The theory about lists of distinct elements (distinct lists) is available in [94] (we used the 2015 version that is available with Isabelle/HOL).

This Appendix is organized as follows: (i) Appendix A.1 presents the base lemmas and theorems for sliceable types; (ii) sublists (sliceable distinct lists) are shown in Appendix A.2; (iii) algebraic definitions and laws of the ATF are shown in Appendix A.3, and (iv) proofs using the denotational semantics of sets of distinct lists are shown in Appendix A.4.

A.1 Sliceable

In this section we present a class to express sub-structures for a data type, and laws over such a class. For example, for lists, *sliceable* defines operators and theorems to obtain sublists.

```
class sliceable =
  fixes slice :: "'a ⇒ nat ⇒ nat ⇒ 'a" ("(3_†_.._)"  [80,80,80] 80)
  fixes size :: "'a ⇒ nat" ("(1#_)" 65)
  fixes empty_inter :: "'a ⇒ 'a ⇒ bool"
  fixes disjoint :: "'a ⇒ bool"

  assumes slice_none: "x†0..(#x) = x"
  assumes empty_seq_inter [simp]:
    "disjoint x ⟹ c ≤ k ⟹ empty_inter (x†0..c) (x†k..(#x))"
  assumes size_slice: "size (x†i..j) = max 0 ((min j (size x))-i)"
  assumes slice_slice: "(x†i..j)†a..b = x†(i+a)..(min j (i+b))"
  assumes disjoint_slice_suc:
    "disjoint x ⟹ i≠j ⟹ i < (#x) ⟹ j < (#x) ⟹
    x†i..(Suc i) ≠ x†j..(Suc j)"
  assumes disjoint_slice[simp]: "disjoint x ⟹ disjoint (x†i..j)"
  assumes forall_slice_implies_eq: "(#x) = (#y) ∧ (∀i j. (x†i..j) =
    (y†i..j)) ⟷ (x = y)"

notation (latex output) slice  ("(3_ [_.._])" [80,80,80] 80)
```

Teste $x_{[i..j]}$

```
definition slice_right :: "'a::sliceable ⇒ nat ⇒ 'a" ("(2_†.._)" [80,80] 80)
where "slice_right x i = x†0..i"

notation ("latex") slice_right  ("(2_ [.._])" [80,80] 80)
```

```
definition slice_left :: "'a::sliceable ⇒ nat ⇒ 'a" ("(2_†_..)" [80,80] 80)
where "x†i.. = x†i..(# x)"

notation ("latex") slice_left  ("(2_ [_..])" [80,80] 80)
```

A.1.1 Disjoint elements and sliceable

```
lemma (in sliceable) slice_right_disjoint[simp]: "disjoint xs ⟹
  disjoint (slice_right xs i)"
unfolding slice_right_def
by simp
```

The notation for $x_{[..i]}$ is $x_{[..i]}$

```
lemma (in sliceable) slice_left_disjoint[simp]: "disjoint xs ⟹
  disjoint (xs†i..)"
unfolding slice_left_def
by simp
```

A.1.2 n-th element in a sliceable

```
abbreviation sliceable_nth :: "'a::sliceable ⇒ nat ⇒ 'a"
where
"sliceable_nth l i ≡ l†i..(Suc i)"
```

A.1.3 Theorems for sliceable

```
theorem (in sliceable) empty_seq_inter_eq [simp]:
  "disjoint x ⟹ empty_inter (x†..i) (x†i..)"
by (simp add: slice_right_def slice_left_def)
```

```
theorem (in sliceable) empty_seq_sliced_inter [simp]:
  "disjoint x ⟹ b ≤ i ⟹ j ≤ a ⟹ i ≤ j ⟹ a ≤ size x ⟹
    empty_inter (x†b..i) (x†j..a)"
proof-
  let ?l = "x†b..a"
  assume lt0: "i ≤ j"
  assume lt1: "j ≤ a"
  assume lt2: "b ≤ i"
  assume lt3: "a ≤ size x"
  assume lt4: "disjoint x"
  have blta: "b ≤ a" using lt0 lt1 lt2 by simp
  have ilta: "i ≤ a" using lt0 lt1 by simp
  hence 2: "empty_inter (?l†0..(i-b)) (?l†(j-b)..(#?l))"
    using lt0 lt4 disjoint_slice by simp
  hence "empty_inter ((x†b..a)†0..(i-b)) ((x†b..a)†(j-b)..(#?l))" by simp
  hence 3: "empty_inter (x†b..i) ((x†b..a)†(j-b)..(#(x†b..a)))" using ilta lt2
    by (simp add: slice_slice min_absorb2)
  hence 3: "empty_inter (x†b..i) (x†j..a)"
    using blta lt0 lt2 lt3
    by (auto simp add: size_slice slice_slice min_def)
  thus ?thesis by simp
qed
```

```
theorem distinct_slice_lte_inter_empty[simp]:
  "distinct l ⟹ i ≤ j ⟹
    set (take i (drop 0 l))
    ∩ set (take (length l-i) (drop i l)) = {}"
by (simp add: set_take_disj_set_drop_if_distinct )
```

```
lemma (in sliceable) size_slice_right_absorb: "(#(l†..i)) = min i (#l)"
by (simp add: slice_right_def sliceable_class.size_slice)
```

```
lemma (in sliceable) size_slice_left_absorb: "(#(l†i..)) = (#l)-i"
by (simp add: slice_left_def sliceable_class.size_slice)
```

```
corollary (in sliceable) slice_right_slice_left_absorb: "(l†..i)†j.. = l†j..i"
unfolding slice_left_def slice_right_def
by (metis (mono_tags, hide_lams) add.left_neutral add.right_neutral max_0L
  min.left_idem size_slice_right_absorb slice_right_def
  sliceable_class.size_slice sliceable_class.slice_none
  sliceable_class.slice_slice)
```

```
corollary (in sliceable) slice_right_slice_left_absorb_empty:
  "i ≤ j ⟹ (#((l†..i)†j..)) = 0"
by (simp add: size_slice_left_absorb size_slice_right_absorb)

corollary (in sliceable) slice_left_slice_right_absorb:
  "(l†i..)†..j = l†i..(i+j)"
unfolding slice_left_def slice_right_def
proof -
  have "(l†i..(#l))†0..j = (l†0..(#l))†i..(i + j)"
    by (simp add: sliceable_class.slice_slice)
  thus "(l†i..(#l))†0..j = l†i..(i + j)"
    by (simp add: sliceable_class.slice_none)
qed

corollary (in sliceable) slice_right_slice_right_absorb:
  "(l†..i)†..j = (l†..(min i j))"
unfolding slice_left_def slice_right_def
by (simp add: sliceable_class.slice_slice)

corollary (in sliceable) slice_left_slice_left_absorb:
  "(l†i..)†j.. = l†(i+j).."
unfolding slice_left_def slice_right_def
by (simp add: sliceable_class.slice_slice sliceable_class.size_slice
  min_absorb1)

corollary (in sliceable) slice_slice_right_absorb:
  "(l†i..j)†..b = l†i..(min j (i+b))"
unfolding slice_left_def slice_right_def
by (simp add: add.commute sliceable_class.slice_slice)

corollary (in sliceable) slice_slice_left_absorb:
  "(l†i..j)†a.. = l†(i+a)..j"
unfolding slice_left_def slice_right_def
by (metis (mono_tags, hide_lams) add.assoc diff_diff_left max_0L
  slice_left_def slice_left_slice_right_absorb slice_right_def
  slice_slice_right_absorb sliceable_class.size_slice
  sliceable_class.slice_none sliceable_class.slice_slice)

corollary (in sliceable) slice_left_slice_absorb:
  "(l†i..)†a..b = l†(i+a)..(i+b)"
unfolding slice_left_def slice_right_def
by (metis (mono_tags, lifting) slice_left_slice_right_absorb slice_right_def
  slice_right_slice_left_absorb slice_slice_left_absorb
  sliceable_class.slice_none)

corollary (in sliceable) slice_right_slice_absorb:
  "(l†..j)†a..b = l†a..(min j b)"
unfolding slice_left_def slice_right_def
by (simp add: sliceable_class.slice_slice)



lemmas (in sliceable) slice_slice_simps =
  slice_left_slice_left_absorb slice_left_slice_right_absorb
  slice_right_slice_left_absorb slice_right_slice_right_absorb slice_slice
  slice_slice_right_absorb slice_slice_left_absorb slice_left_slice_absorb
  slice_right_slice_absorb

lemmas (in sliceable) size_slice_defs =
  size_slice size_slice_left_absorb size_slice_right_absorb

lemma (in sliceable) slice_f_min_neutral:
  "(P (l†i..(min f k)) ∧ f ≤ k) ⟷ (P (l†i..f) ∧ f ≤ k)"
by linarith

lemma (in sliceable) slice_i_min_neutral:
  "(P (l†(min i k)..f) ∧ i ≤ k) ⟷ (P (l†i..f) ∧ i ≤ k)"
by linarith

lemma (in sliceable) slice_i_min_neutral_lt:
  "(P (l†(min k i)..f) ∧ i < k) ⟷ (P (l†i..f) ∧ i < k)"
by linarith

lemma (in sliceable) slice_foral_i_min_neutral:
  "(∀ i f . P (l†(min i k)..f) ∧ i ≤ k) ⟷ (∀ i f . P (l†i..f) ∧ i ≤ k)"
using not_less by auto

lemma (in sliceable) slice_f_max_neutral:
  "(P (l†i..(max f k)) ∧ f ≥ k) ⟷ (P (l†i..f) ∧ f ≥ k)"
by (metis max.orderE)

lemma (in sliceable) slice_i_max_neutral:
  "(P (l†(max i k)..f) ∧ i ≥ k) ⟷ (P (l†i..f) ∧ i ≥ k)"
by (metis max.orderE)
```

```
lemma (in sliceable) empty_slice[simp]: "i ≤ j ⟹ (#(l†j..i)) = 0"
using local.size_slice by auto

corollary (in sliceable) forall_disjoint_slice_suc:
  "∀ i j . (disjoint x ∧ i≠j ∧ i < (#x) ∧ j < (#x)) ⟶
    (x†i..(Suc i) ≠ x†j..(Suc j))"
by (simp add: local.disjoint_slice_suc)

lemma (in sliceable) empty_slice_none:
  "(#x) = 0 ⟹ (#(x†i..j)) = 0"
by (simp add: size_slice)

corollary (in sliceable) empty_slice_right_none:
  "(#x) = 0 ⟹ (#(x†..j)) = 0"
by (simp add: slice_right_def sliceable_class.empty_slice_none)

corollary (in sliceable) empty_slice_left_none:
  "(#x) = 0 ⟹ (#(x†i..)) = 0"
by (simp add: slice_left_def sliceable_class.empty_slice_none)
```

A.2 Sliceable distinct lists

The following is the instantiation of the sliceable class for the dlist type.

```
instantiation dlist :: (type) sliceable
begin

definition
  "l†i..f = Dlist (take (max 0 (f-i)) (drop i (list_of_dlist l)))"

definition
  "size l = length (list_of_dlist l)"

definition
  "empty_inter l k =
  ((set (list_of_dlist l)) ∩ (set (list_of_dlist k)) = {})"

definition
  "disjoint l = distinct (list_of_dlist l)"

lemma list_of_dlist_slice :
  "list_of_dlist (l†i..f) = take (max 0 (f-i)) (drop i (list_of_dlist l))"
unfolding slice_dlist_def
by simp

lemma Dlist_slice_inverse :
  "list_of_dlist (Dlist (take (max 0 (c-i)) (drop i (list_of_dlist x))))
  = (take (max 0 (c-i)) (drop i (list_of_dlist x)))"
by simp

lemma Dlist_empty_seq_inter: "c ≤ k ⟹
  (
  set (take c (list_of_dlist x)) ∩
  set (drop k (list_of_dlist x))
  ) = {}"
by (simp add: set_take_disj_set_drop_if_distinct)

lemma Dlist_forall_slice_eq1:
  "(∀i f. (Dlist (take (max 0 (f-i)) (drop i (list_of_dlist l1))) =
  Dlist (take (max 0 (f-i)) (drop i (list_of_dlist l2))))) ⟹
  l1 = l2"
by (metis (mono_tags, hide_lams) Dlist_list_of_dlist
  Sliceable_dlist.list_of_dlist_slice drop_0 drop_take max_0L take_equalityI)

lemma Dlist_forall_slice_eq:
  "l1 = l2 ⟷
  (∀i f. (Dlist (take (max 0 (f-i)) (drop i (list_of_dlist l1))) =
  Dlist (take (max 0 (f-i)) (drop i (list_of_dlist l2)))))"
using Dlist_forall_slice_eq1 by blast

lemma distinct_list_take_1_uniqueness:
  "distinct l ⟹ i≠j ⟹ i < length l ⟹ j < length l ⟹
    take 1 (drop i l) ≠ take 1 (drop j l)"
by (simp add: hd_drop_conv_nth nth_eq_iff_index_eq take_Suc)

lemmas list_of_dlist_simps = slice_left_def slice_right_def slice_dlist_def
  size_dlist_def disjoint_dlist_def empty_inter_dlist_def Dlist_slice_inverse

instance proof
  fix l::"’a dlist"
  show "l†0..(#l) = l" by (simp add: dlist_eqI list_of_dlist_slice size_dlist_def)

  fix l::"’a dlist"
  show "disjoint l" by (simp add: disjoint_dlist_def)
  next
  fix l::"'a dlist" and c::nat and k
  assume "c ≤ k"
  thus "empty_inter (l†0..c) (l†k..(#l))"
  by (simp add: size_dlist_def empty_inter_dlist_def
    set_take_disj_set_drop_if_distinct list_of_dlist_slice )
  next
  fix l::"'a dlist" and i and j and a and b
  show "size (l†i..j) = max 0 (min j (#l) - i)"
  proof (cases "j ≤ #l")
    case True
    assume "j ≤ #l"
    thus ?thesis
      by (metis (no_types, hide_lams) list_of_dlist_simps(7) size_dlist_def
        drop_take length_drop length_take list_of_dlist_simps(3) max_0L
        min.commute)
    next
    case False
    assume "¬ (j ≤ #l)"
    hence "j > #l" by simp
    thus ?thesis
      by (metis (no_types, lifting) list_of_dlist_simps(3)
        list_of_dlist_simps(7) size_dlist_def length_drop length_take max_0L
        min.commute min_diff)
  qed
  next
  fix l::"'a dlist" and i and j and a and b
  show "(l†i..j)†a..b = l†(i + a)..(min j (i + b))"
    proof -
      have f1: "∀n. max (0::nat) n = n"
        by (meson max_0L)
      hence "take b (take (max 0 (j - i)) (drop i (list_of_dlist l))) = drop i (take
(i + b) (take j (list_of_dlist l)))"
        by (metis (no_types) diff_add_inverse drop_take)
      hence "take (max 0 (b - a)) (drop a (list_of_dlist (l†i..j))) = drop a (drop
i (take (min (i + b) j) (list_of_dlist l)))"
        using f1 by (metis Sliceable_dlist.list_of_dlist_slice drop_take take_take)
      thus ?thesis
        using f1 by (metis (no_types) add.commute drop_drop drop_take list_of_dlist_simps(3)
min.commute)
    qed
  next
  fix l::"'a dlist" and i and j
  assume "disjoint l" "i≠j" "i < (#l)" "j < (#l)"
  hence "take 1 (drop i (list_of_dlist l)) ≠
    take 1 (drop j (list_of_dlist l))"
    using distinct_list_take_1_uniqueness size_dlist_def by auto
  hence "take (Suc i - i) (drop i (list_of_dlist l)) ≠
    take (Suc j - j) (drop j (list_of_dlist l))"
    by simp
  hence "take (max 0 (Suc i - i)) (drop i (list_of_dlist l)) ≠
    take (max 0 (Suc j - j)) (drop j (list_of_dlist l))"
    by simp
  thus "l†i..Suc i ≠ l†j..Suc j"
  by (metis list_of_dlist_slice)
  next
  fix l1::"'a dlist" and l2::"'a dlist"
  show "(#l1) = (#l2) ∧ (∀i j. l1†i..j = l2†i..j) ⟷ (l1 = l2)"
    using Dlist_forall_slice_eq
    by (metis Sliceable_dlist.list_of_dlist_slice)
qed

end
```

A.2.1 Properties of sliceable distinct lists

In the following we present lemmas, corollaries and theorems about sliceable distinct lists.

```
abbreviation dlist_nth :: "'a dlist ⇒ nat ⇒ 'a"
where
"dlist_nth l i ≡ (list_of_dlist (sliceable_nth l i))!0"

theorem set_slice :
  "set (list_of_dlist l) =
    set (list_of_dlist (l†..i)) ∪ set (list_of_dlist (l†i..))"
unfolding slice_dlist_def slice_right_def slice_left_def size_dlist_def
apply (simp add: list_of_dlist_inject)
by (metis append_take_drop_id set_append)

theorem take_slice_right: "take n (list_of_dlist l) = list_of_dlist (l†..n)"
unfolding slice_right_def slice_dlist_def
by (metis Dlist_slice_inverse drop_0 max_0L minus_nat.diff_0)
```

```
theorem slice_right_cons: "distinct (x # xs) ⟹
  (Dlist (x # xs))†..(Suc n) = Dlist (x # (list_of_dlist ((Dlist xs)†..n)))"
unfolding slice_right_def slice_dlist_def
by (simp add: distinct_remdups_id)

theorem slice_append:
  "∀n. Dlist ((list_of_dlist (l†..n)) @ (list_of_dlist (l†n..))) = l"
unfolding size_dlist_def slice_left_def slice_right_def
by (simp add: list_of_dlist_inverse list_of_dlist_slice )

theorem slice_append_mid:
"∀i s e. s ≤ i ∧ i ≤ e ⟶
  ((list_of_dlist (l†s..i)) @ (list_of_dlist (l†i..e))) =
    list_of_dlist (l†s..e)"
unfolding size_dlist_def slice_left_def slice_right_def list_of_dlist_slice
by (smt Nat.diff_add_assoc2 drop_drop le_add_diff_inverse
  le_add_diff_inverse2 max_0L take_add)

theorem slice_append_3:
"∀i j. i ≤ j ⟶
  ((list_of_dlist (l†..i)) @
    (list_of_dlist (l†i..j)) @ (list_of_dlist (l†j..))) = list_of_dlist l"
unfolding size_dlist_def slice_left_def slice_right_def list_of_dlist_slice
by (metis append_assoc append_take_drop_id drop_0 le_add_diff_inverse
  length_drop max.cobounded2 max_0L minus_nat.diff_0 take_add take_all)

theorem distinct_slice_lte_inter_empty[simp]:
  "i ≤ j ⟹ set (list_of_dlist (l†..i)) ∩ set (list_of_dlist (l†j..)) = {}"
unfolding size_dlist_def slice_left_def slice_right_def
by (simp add: Dlist_empty_seq_inter list_of_dlist_slice )

corollary distinct_slice_inter_empty [simp]:
  "set (list_of_dlist (l†..i)) ∩ set (list_of_dlist (l†i..)) = {}"
by simp

corollary distinct_slice_lt_inter_empty [simp]:
  "i < j ⟹ set (list_of_dlist (l†..i)) ∩ set (list_of_dlist (l†j..)) = {}"
by simp

corollary distinct_slice_diff1:
  "set (list_of_dlist (l†..i)) - set (list_of_dlist (l†i..)) =
    set (list_of_dlist (l†..i))"
by (simp add: Diff_triv)

corollary distinct_slice_diff2:
  "set (list_of_dlist (l†i..)) - set (list_of_dlist (l†..i)) =
    set (list_of_dlist (l†i..))"
using distinct_slice_diff1 by fastforce
```

```
theorem distinct_in_set_slice1_not_in_slice2:
  "i ≤ j ⟹
  x ∈ set (list_of_dlist (l†..i)) ∧ x ∈ set (list_of_dlist (l†j..)) ⟹
  False"
using distinct_slice_lte_inter_empty by fastforce

corollary distinct_in_set_slice1_implies_not_in_slice2:
  "i ≤ j ⟹ x ∈ set (list_of_dlist (l†..i)) ⟹
  x ∈ set (list_of_dlist (l†j..)) ⟹ False"
by (meson distinct_in_set_slice1_not_in_slice2)

lemma exists_sublist_or_not_sublist [simp]: "∃i. l†..i ∈ T ∨ l†i.. ∉ T"
unfolding slice_right_def slice_left_def
by auto

lemma forall_slice_left_implies_exists [simp]:
  "∀ i . l†i.. ∈ S ⟹ ∃ i . l†(Suc i).. ∈ S"
unfolding slice_right_def slice_left_def
by (simp add: slice_dlist_def)

lemma forall_slice_right_implies_exists [simp]:
  "∀ i . l†..i ∈ S ⟹ ∃ i . l†..(i-1) ∈ S"
unfolding slice_right_def slice_left_def
by auto
```

```
lemma take_Suc_Cons_hd_tl: "length l > 0 ⟹
  take (Suc n) l = hd l # (take n (tl l))"
apply (induct l)
by auto
```

```
corollary take_Suc_Cons_hd_tl_singleton:
  "length l > 0 ⟹ take (Suc 0) l = [hd l]"
apply (induct l)
by auto

lemma take_drop_suc: "i < length l ⟹ length l > 0 ⟹
  take (max 0 ((Suc i) - i)) (drop i l) = [l!i]"
by (metis (no_types, lifting) Suc_diff_Suc Suc_eq_plus1_left add.commute
  append_eq_append_conv cancel_comm_monoid_add_class.diff_cancel
  hd_drop_conv_nth lessI max_0L numeral_1_eq_Suc_0 numeral_One take_add
  take_hd_drop)

lemma slice_right_take:"l†..i = Dlist (take i (list_of_dlist l))"
unfolding slice_right_def slice_dlist_def
by auto

lemma slice_left_drop: "l†i.. = Dlist (drop i (list_of_dlist l))"
unfolding slice_left_def slice_dlist_def size_dlist_def
by auto

lemma take_one_singleton_hd: "l ≠ [] ⟹ take (Suc 0) l = [hd l]"
apply (induct l, simp)
by auto

lemma take_one_singleton_nth: "l ≠ [] ⟹ take (Suc 0) l = [l!0]"
apply (induct l, simp)
by auto

lemma take_one_drop_n_append_singleton_nth:
  "ys ≠ [] ⟹ take 1 (drop (length xs) (xs @ ys)) =
  [(xs @ ys)!(length xs)]"
by (induct xs, auto simp add: take_one_singleton_nth)

lemma append_length_nth_hd: "ys ≠ [] ⟹ [(xs @ ys)!(length xs)] = [hd ys]"
by (induct ys, auto)

lemma take_one_drop_n_singleton_nth: "l ≠ [] ⟹ n < length l ⟹
  take 1 (drop n l) = [l!n]"
proof-
  assume 0: "l ≠ []"
  assume 1: "n < length l"
  obtain xs where "xs = take n l" by simp
  obtain ys where "ys = drop n l" by simp
  have "take 1 (drop n l) = take 1 (drop (length xs) (xs @ ys))" using 0 1
    by (simp add: `ys = drop n l`)
  also have "... = [(xs @ ys)!(length xs)]" using 0 1
    by (metis `ys = drop n l` drop_eq_Nil not_le
      take_one_drop_n_append_singleton_nth)
  also have "... = [l!(length xs)]"
    by (simp add: `xs = take n l` `ys = drop n l`)
  finally show ?thesis using 0 1
    by (simp add: hd_drop_conv_nth take_one_singleton_hd)
qed

lemma slice_singleton: "(list_of_dlist l) ≠ [] ⟹ i < (#l) ⟹
  list_of_dlist (l†i..(Suc i)) = [(list_of_dlist l)!i]"
by (metis list_of_dlist_slice length_greater_0_conv size_dlist_def
  take_drop_suc)

lemma slice_right_zero_eq_empty: "list_of_dlist (l†..0) = []"
by (simp add: slice_right_def slice_dlist_def)

lemma slice_left_size_eq_empty: "list_of_dlist (l†(#l)..) = []"
by (simp add: slice_left_def slice_dlist_def )

lemma slice_right_singleton_eq_element: "list_of_dlist l ≠ [] ⟹
  list_of_dlist (l†..1) = [(list_of_dlist l)!0]"
by (metis One_nat_def take_one_singleton_nth take_slice_right)

lemma slice_left_singleton_eq_element: "list_of_dlist l ≠ [] ⟹
  list_of_dlist (l†((#l)-1)..) = [(list_of_dlist l)!((#l)-1)]"
by (metis (no_types, lifting) Cons_nth_drop_Suc list_of_dlist_slice
  Suc_diff_Suc Suc_leI diff_Suc_eq_diff_pred diff_less drop_0 drop_all
  drop_take length_greater_0_conv max_0L minus_nat.diff_0 size_dlist_def
  slice_left_def slice_none zero_less_one)

lemma dlist_empty_slice[simp]: "i ≤ j ⟹ (l†j..i) = Dlist []"
by (simp add: slice_dlist_def)

lemma dlist_append_extreme_left:
  "i≤j ⟹ list_of_dlist (l†..j) =
  (list_of_dlist (l†..i)) @ (list_of_dlist (l†i..j))"
by (metis list_of_dlist_slice le_add_diff_inverse max_0L take_add
  take_slice_right)
```

```
lemma dlist_append_extreme_right:
  "i≤j ⟹ list_of_dlist (l†i..) =
  (list_of_dlist (l†i..j)) @ (list_of_dlist (l†j..))"
unfolding list_of_dlist_slice slice_left_def slice_right_def
by (metis append_take_drop_id drop_drop le_add_diff_inverse2 length_drop
  max.cobounded2 max_0L size_dlist_def take_all)

lemma dlist_disjoint[simp]: "disjoint (l::'a dlist)"
by (simp add: disjoint_dlist_def)

lemma dlist_member_suc_nth1:
  "x ∈ set (list_of_dlist(l†i..(Suc i))) ⟹ x = (list_of_dlist l)!i"
proof-
  assume 0: "x ∈ set (list_of_dlist (l†i..(Suc i)))"
  obtain rl where 1:"rl = list_of_dlist l" by blast
  hence "x ∈ set (take (max 0 (Suc i - i)) (drop i rl))"
    using 0 by (metis list_of_dlist_slice )
  hence "x ∈ set (take 1 (drop i rl))" by simp
  hence "x = rl!i"
    by (metis drop_Nil drop_all empty_iff list.inject list.set(1)
      list.set_cases not_less take_Nil take_one_drop_n_singleton_nth)
  thus ?thesis using 1 by simp
qed

lemma dlist_member_suc_nth2:
  "i < (#l) ⟹ x = (list_of_dlist l)!i ⟹
  x ∈ set (list_of_dlist (l†i..(Suc i)))"
unfolding size_dlist_def slice_dlist_def
by (metis Dlist_slice_inverse drop_Nil drop_eq_Nil leD length_greater_0_conv
  list.set_intros(1) take_drop_suc)

lemma dlist_member_suc_nth: "i < (#l) ⟹
  (x = (list_of_dlist l)!i) ⟷ (x ∈ set (list_of_dlist (l†i..(Suc i))))"
using dlist_member_suc_nth1 dlist_member_suc_nth2
by fastforce

corollary not_dlist_member_empty[simp]:
  "¬ Dlist.member (Dlist.empty) v"
  "¬ (Dlist.member (Dlist []) v)"
by (simp add: Dlist.member_def Dlist.empty_def List.member_def)+
lemma dlist_empty_slice_none: "(Dlist.empty†i..j) = Dlist.empty"
by (simp add: Dlist.empty_def slice_dlist_def)

corollary dlist_empty_slice_right_none: "(Dlist.empty†..j) = Dlist.empty"
by (simp add: dlist_empty_slice_none slice_right_def)

corollary dlist_empty_slice_left_none: "(Dlist.empty†i..) = Dlist.empty"
by (simp add: dlist_empty_slice_none slice_left_def)

lemma dlist_member_slice_empty_none:
  "¬ (Dlist.member (Dlist.empty†i..j) v)"
by (auto simp add: slice_dlist_def)

corollary dlist_member_slice_right_empty_none[simp]:
  "¬ (Dlist.member (Dlist.empty†..j) v)"
by (simp add: slice_right_def dlist_empty_slice_none)

corollary dlist_member_slice_left_empty_none[simp]:
  "¬ (Dlist.member (Dlist.empty†i..) v)"
by (simp add: slice_left_def dlist_empty_slice_none)

lemma dlist_member_slice_member_dlist:
  "∃ i j. Dlist.member (dl†i..j) v ⟹ Dlist.member dl v"
unfolding Dlist.member_def List.member_def slice_dlist_def
using in_set_dropD in_set_takeD by fastforce

corollary dlist_member_slice_right_member_dlist:
  "∃j. Dlist.member (dl†..j) v ⟹ Dlist.member dl v"
by (metis dlist_member_slice_member_dlist slice_right_def)

corollary dlist_member_slice_left_member_dlist:
  "∃i. Dlist.member (dl†i..) v ⟹ Dlist.member dl v"
by (metis dlist_member_slice_member_dlist slice_left_def)

lemma sliceable_nth_member1:
  "sliceable_nth dl i = Dlist [v] ⟹ Dlist.member dl v"
by (metis Dlist.member_def distinct_remdups_id distinct_singleton
  dlist_member_slice_member_dlist in_set_member list.set_intros(1) list_of_dlist_Dlist)

corollary sliceable_nth_member:
  "∃i. sliceable_nth dl i = Dlist [v] ⟹ Dlist.member dl v"
by (auto simp add: sliceable_nth_member1)
```

```
lemma sliceable_nth_member_iff:
  "(∃i. sliceable_nth dl i = Dlist [v]) ⟷ Dlist.member dl v"
apply (rule iffI, simp add: sliceable_nth_member)
by (metis Dlist.member_def empty_iff empty_set in_set_conv_nth in_set_member
  list_of_dlist_slice size_dlist_def slice_dlist_def slice_singleton)
```

A.3 Algebra of Temporal Faults

In the following we present the algebraic laws for the ATF.

A.3.1 Basic ATF operators and $\textbf{tempo}_1$

```
class temporal_faults_algebra_basic = boolean_algebra  +
  fixes neutral :: "'a"
  fixes xbefore :: "'a ⇒ 'a ⇒ 'a"
  fixes tempo1 :: "'a ⇒ bool"
  assumes xbefore_bot_1: "xbefore bot a = bot"
  assumes xbefore_bot_2: "xbefore a bot = bot"
  assumes xbefore_neutral_1: "tempo1 a ⟹ xbefore neutral a = a"
  assumes xbefore_neutral_2: "tempo1 a ⟹ xbefore a neutral = a"
  assumes xbefore_not_idempotent: "tempo1 a ⟹ xbefore a a = bot"
  assumes inf_tempo1: "⟦tempo1 a; tempo1 b⟧ ⟹ tempo1 (inf a b)"
  assumes xbefore_not_sym:
    "⟦tempo1 a; tempo1 b⟧ ⟹ (xbefore a b) ≤ -(xbefore b a)"
```

A.3.2 Definition of associativity of XBefore

```
class temporal_faults_algebra_assoc = temporal_faults_algebra_basic +
  assumes xbefore_assoc: "⟦tempo1 a; tempo1 b; tempo1 c⟧ ⟹
    xbefore (xbefore a b) c = xbefore a (xbefore b c)"
```

A.3.3 Equivalences in the ATF and properties

```
class temporal_faults_algebra_equivs = temporal_faults_algebra_assoc +
  fixes independent_events :: "'a ⇒ 'a ⇒ bool"
  fixes tempo2 :: "'a ⇒ bool"
  fixes tempo3 :: "'a ⇒ bool"
  fixes tempo4 :: "'a ⇒ bool"
  assumes xbefore_inf_equiv_bot:
    "⟦tempo1 a; tempo1 b⟧ ⟹ inf (xbefore a b) (xbefore b a) = bot"
  assumes xbefore_sup_equiv_inf:
    "independent_events a b ⟹ ⟦tempo1 a; tempo1 b⟧ ⟹
    ⟦tempo2 a; tempo2 b⟧ ⟹ ⟦tempo3 a; tempo3 b⟧ ⟹ ⟦tempo4 a; tempo4 b⟧ ⟹
    sup (xbefore a b) (xbefore b a) = inf a b"
  assumes sup_tempo2: "⟦tempo2 a; tempo2 b⟧ ⟹ tempo2 (sup a b)"
  assumes inf_tempo3: "⟦tempo3 a; tempo3 b⟧ ⟹ tempo3 (inf a b)"
  assumes sup_tempo4: "⟦tempo4 a; tempo4 b⟧ ⟹ tempo4 (sup a b)"
```

A.3.4 XBefore transitivity

```
class temporal_faults_algebra_trans = temporal_faults_algebra_equivs +
  assumes xbefore_trans:
    "⟦tempo1 a; tempo1 b; tempo1 c⟧ ⟹ ⟦tempo2 a; tempo2 b; tempo2 c⟧ ⟹
    less_eq (inf (xbefore a b) (xbefore b c)) (xbefore a c)"
  assumes inf_xbefore_trans: "⟦ tempo1 b; tempo3 b ⟧ ⟹
    inf (xbefore a b) (xbefore b c) = xbefore (xbefore a b) c"
```

A.3.5 Mixed operators in ATF

```
class temporal_faults_algebra_mixed_ops = temporal_faults_algebra_trans +
  assumes xbefore_sup_1:
    "xbefore (sup a b) c = sup (xbefore a c) (xbefore b c)"
  assumes xbefore_sup_2:
    "xbefore a (sup b c) = sup (xbefore a b) (xbefore a c)"
  assumes not_xbefore: "
    independent_events a b ⟹
    ⟦tempo1 a; tempo1 b⟧ ⟹
    ⟦tempo2 a; tempo2 b⟧ ⟹
    ⟦tempo3 a; tempo3 b⟧ ⟹
    ⟦tempo4 a; tempo4 b⟧ ⟹
    - (xbefore a b) = sup (sup (- a) (- b)) (xbefore b a)"
  assumes inf_xbefore_equiv_sups_xbefore: "tempo2 a ⟹
    inf a (xbefore b c) = sup (xbefore (inf a b) c) (xbefore b (inf a c))"
  assumes not_1_xbefore_equiv: "⟦tempo1 a; tempo2 b ⟧ ⟹ xbefore (- a) b = b"
  assumes not_2_xbefore_equiv: "⟦tempo1 b; tempo2 a ⟧ ⟹ xbefore a (- b) = a"

class temporal_faults_algebra = temporal_faults_algebra_mixed_ops
```

A.3.6 Theorems in the context of ATF

The following theorems are valid for ATF. They are valid for any instantiation of the ATF class as, for example, for the sets of distinct lists type.

```
context temporal_faults_algebra
begin


theorem xbefore_inf_1:
  "independent_events a b ⟹ ⟦tempo1 a; tempo1 b⟧ ⟹
  ⟦tempo2 a; tempo2 b⟧ ⟹ ⟦tempo3 a; tempo3 b⟧ ⟹ ⟦tempo4 a; tempo4 b⟧ ⟹
  xbefore (inf a b) c =
    sup (xbefore (xbefore a b) c) (xbefore (xbefore b a) c)"
proof-
  assume "independent_events a b" "tempo1 a" "tempo1 b"
  "tempo2 a" "tempo2 b" "tempo3 a" "tempo3 b" "tempo4 a" "tempo4 b"
  hence "xbefore (inf a b) c = xbefore (sup (xbefore a b) (xbefore b a)) c"
    by (simp add: xbefore_sup_equiv_inf)
  thus ?thesis by (simp add: xbefore_sup_1)
qed


theorem xbefore_inf_2:
  "independent_events b c ⟹ ⟦tempo1 b; tempo1 c⟧ ⟹
  ⟦tempo2 b; tempo2 c⟧ ⟹ ⟦tempo3 b; tempo3 c⟧ ⟹ ⟦tempo4 b; tempo4 c⟧ ⟹
  xbefore a (inf b c) =
    sup (xbefore a (xbefore b c)) (xbefore a (xbefore c b))"
proof-
  assume "independent_events b c" "tempo1 b" "tempo1 c" "tempo2 b" "tempo2 c"
  "tempo3 b" "tempo3 c" "tempo4 b" "tempo4 c"
  hence "xbefore a (inf b c) = xbefore a (sup (xbefore b c) (xbefore c b))"
    by (simp add: xbefore_sup_equiv_inf)
  thus ?thesis by (simp add: xbefore_sup_2)
qed


lemma xbefore_sup_absorb_1b:
  "independent_events a b ⟹ ⟦tempo1 a; tempo1 b⟧ ⟹
    ⟦tempo2 a; tempo2 b⟧ ⟹ ⟦tempo3 a; tempo3 b⟧ ⟹ ⟦tempo4 a; tempo4 b⟧ ⟹
    sup (xbefore b a) a = a"
by (metis inf_le1 order_trans sup.absorb2 sup.cobounded2
  xbefore_sup_equiv_inf)


lemma xbefore_sup_absorb_2:
  "independent_events a b ⟹ ⟦tempo1 a; tempo1 b⟧ ⟹
```

```
    ⟦tempo2 a; tempo2 b⟧ ⟹ ⟦tempo3 a; tempo3 b⟧ ⟹ ⟦tempo4 a; tempo4 b⟧ ⟹
    sup a (xbefore a b) = a"
by (metis dual_order.trans inf.cobounded1 sup.absorb1 sup.cobounded1
  xbefore_sup_equiv_inf)

corollary xbefore_sup_absorb_1:
  "independent_events a b ⟹ ⟦tempo1 a; tempo1 b⟧ ⟹
    ⟦tempo2 a; tempo2 b⟧ ⟹ ⟦tempo3 a; tempo3 b⟧ ⟹ ⟦tempo4 a; tempo4 b⟧ ⟹
    sup (xbefore a b) a = a"
proof-
  assume 0: "independent_events a b" "tempo1 a" "tempo1 b" "tempo2 a"
  "tempo2 b" "tempo3 a" "tempo3 b" "tempo4 a" "tempo4 b"
  hence "sup a (xbefore a b) = sup (xbefore a b) a"
    by (simp add: sup.commute)
  thus ?thesis using 0 by (simp add: xbefore_sup_absorb_2)
qed

corollary xbefore_sup_absorb_2b:
  "independent_events a b ⟹ ⟦tempo1 a; tempo1 b⟧ ⟹
    ⟦tempo2 a; tempo2 b⟧ ⟹ ⟦tempo3 a; tempo3 b⟧ ⟹ ⟦tempo4 a; tempo4 b⟧ ⟹
    sup a (xbefore b a) = a"
proof-
  assume 0: "independent_events a b" "tempo1 a" "tempo1 b" "tempo2 a"
  "tempo2 b" "tempo3 a" "tempo3 b" "tempo4 a" "tempo4 b"
  hence "sup a (xbefore b a) = sup (xbefore b a) a"
    by (simp add: sup.commute)
  thus ?thesis using 0 by (simp add: xbefore_sup_absorb_1b)
qed

theorem xbefore_inf_absorb_1: "independent_events a b ⟹
  ⟦ tempo1 a; tempo1 b ⟧ ⟹
  ⟦ tempo2 a; tempo2 b ⟧ ⟹
  ⟦ tempo3 a; tempo3 b ⟧ ⟹
  ⟦ tempo4 a; tempo4 b ⟧ ⟹
  inf a (xbefore a b) = xbefore a b"
by (simp add: local.inf_absorb2 local.le_iff_sup xbefore_sup_absorb_1)

theorem xbefore_inf_absorb_2: "independent_events a b ⟹
  ⟦ tempo1 a; tempo1 b ⟧ ⟹
  ⟦ tempo2 a; tempo2 b ⟧ ⟹
  ⟦ tempo3 a; tempo3 b ⟧ ⟹
  ⟦ tempo4 a; tempo4 b ⟧ ⟹
  inf a (xbefore b a) = xbefore b a"
by (simp add: local.inf.absorb2 local.sup.absorb_iff1 xbefore_sup_absorb_2b)


lemma inf_xbefore_equiv_sups_xbefore_expanded:
  "independent_events a b ⟹ independent_events a c ⟹
  ⟦tempo1 a; tempo1 b; tempo1 c⟧ ⟹ ⟦tempo2 a; tempo2 b; tempo2 c⟧ ⟹
  ⟦tempo3 a; tempo3 b; tempo3 c⟧ ⟹ ⟦tempo4 a; tempo4 b; tempo4 c⟧ ⟹
    inf a (xbefore b c) =
    sup (sup (xbefore (xbefore a b) c)
      (xbefore (xbefore b a) c))
      (xbefore (xbefore b c) a)"
proof-
  assume "independent_events a b" "independent_events a c"
    "tempo1 a" "tempo1 b" "tempo1 c"
    "tempo2 a" "tempo2 b" "tempo2 c"
    "tempo3 a" "tempo3 b" "tempo3 c"
    "tempo4 a" "tempo4 b" "tempo4 c"
  hence "inf a (xbefore b c) =
    sup (xbefore (inf a b) c) (xbefore b (inf a c))"
    "xbefore (inf a b) c =
    sup (xbefore (xbefore a b) c) (xbefore (xbefore b a) c)"
    "xbefore b (inf a c) =
    sup (xbefore (xbefore b a) c) (xbefore (xbefore b c) a)"
    by (auto simp add: inf_xbefore_equiv_sups_xbefore xbefore_inf_1
      xbefore_inf_2 xbefore_assoc)
  thus ?thesis by (simp add: sup.assoc)
qed


corollary inf_xbefore_equiv_sups_xbefore_1:
  "independent_events a b ⟹ independent_events a c  ⟹
  ⟦tempo1 a; tempo1 b; tempo1 c⟧ ⟹ ⟦tempo2 a; tempo2 b; tempo2 c⟧ ⟹
  ⟦tempo3 a; tempo3 b; tempo3 c⟧ ⟹ ⟦tempo4 a; tempo4 b; tempo4 c⟧ ⟹
  inf a (xbefore b c) =
  sup (sup (xbefore (xbefore a b) c) (xbefore (xbefore b a) c))
    (xbefore (xbefore b c) a)"
by (simp add: inf_xbefore_equiv_sups_xbefore_expanded)


lemma xbefore_sup_compl_inf_absorb1:
  "independent_events a b ⟹ ⟦tempo1 a; tempo1 b⟧ ⟹
    ⟦tempo2 a; tempo2 b⟧ ⟹ ⟦tempo3 a; tempo3 b⟧ ⟹ ⟦tempo4 a; tempo4 b⟧ ⟹
    sup (inf a (-b)) (xbefore a b) = inf a (- (xbefore b a))"
proof -
```

```
  assume a1: "independent_events a b"
  assume a2: "tempo1 a"
  assume a3: "tempo1 b"
  assume a4: "tempo2 a"
  assume a5: "tempo2 b"
  assume a6: "tempo3 a"
  assume a7: "tempo3 b"
  assume a8: "tempo4 a"
  assume a9: "tempo4 b"
  then have f10: "- xbefore a b = sup (sup (- a) (- b)) (xbefore b a)"
    using a8 a7 a6 a5 a4 a3 a2 a1 by (meson local.not_xbefore)
  have f11: "∀a aa ab. inf (a::'a) (sup aa ab) = sup (inf a aa) (inf a ab)"
    using local.distrib_imp2 local.sup_inf_distrib1 by force
  then have f12: "sup bot (xbefore b a) = inf b (sup (- b) (xbefore b a))"
    using a7 a3 by (metis local.inf_compl_bot local.inf_xbefore_trans local.xbefore_neutr
  have f13: "inf (sup (- a) (- (- b))) (sup (- a) (- b)) = sup (- a) (inf b (- b))"
    using local.double_compl local.sup_inf_distrib1 by presburger
  have f14: "inf a (xbefore b a) = xbefore b a"
    using a9 a8 a7 a6 a5 a4 a3 a2 a1 by (meson xbefore_inf_absorb_2)
  have f15: "sup (sup (inf (sup (- a) (- b)) (- a)) (inf (xbefore b a) (- a))) bot
= sup (- a) (inf b (- b))"
    using f10 a9 a8 a7 a6 a5 a4 a3 a2 a1 by (metis (no_types) local.compl_sup local.inf_c
local.inf_sup_distrib2 xbefore_sup_absorb_1)
  have "inf a (- a) = sup (inf (xbefore b a) (- a)) bot"
    using f14 f11 by (metis (no_types) local.compl_inf local.inf_compl_bot)
  then have "sup (- sup (- a) (- (- b))) (- sup (- a) (- b)) = - inf (sup (sup (-
a) (- b)) a) (- a)"
    using f15 f13 by (metis (full_types) local.compl_inf local.inf_sup_distrib2
local.sup_assoc)
  then have "sup (- sup (- a) (- (- b))) (- sup (- a) (- b)) = a"
    by (simp add: local.sup_assoc)
  then show ?thesis
    using f13 f12 f10 by (metis (no_types) local.compl_inf local.compl_sup local.double_c
local.inf_compl_bot local.sup_assoc local.sup_inf_distrib1)
qed

end

end
```

A.4 Denotational semantics for ATF

In the following we present the denotation semantics for ATF in terms of sets of distinct lists.

A.4.1 Formula: distinct lists

The definition of a formula in the ATF is a set of sets of distinct lists (dlist).

typedef *'a formula = "UNIV::'a dlist set set"* **by** *simp*

A.4.1.1 Formula as Boolean algebra

In the following we instantiate the formula as a Boolean algebra and prove that Boolean operators are valid.

instantiation *formula :: (type) boolean_algebra*
begin

definition
"x ⊓ y = Abs_formula (Rep_formula x ∩ Rep_formula y)"

definition
"x ⊔ y = Abs_formula (Rep_formula x ∪ Rep_formula y)"

definition
"⊤ = Abs_formula UNIV"

definition
"⊥ = Abs_formula {}"

definition
"x ≤ y ⟷ Rep_formula x ⊆ Rep_formula y"

definition
"x < y ⟷ Rep_formula x ⊂ Rep_formula y"

definition
"- x = Abs_formula (- (Rep_formula x))"

definition
"x - y = Abs_formula (Rep_formula x - Rep_formula y)"

```
lemma Rep_formula_inf:
  "Rep_formula (x ⊓ y) = Rep_formula x ∩ Rep_formula y"
unfolding inf_formula_def
by (simp add: Abs_formula_inverse Rep_formula)

lemma Rep_formula_sup:
  "Rep_formula (x ⊔ y) = Rep_formula x ∪ Rep_formula y"
unfolding sup_formula_def
by (simp add: Abs_formula_inverse Rep_formula)

lemma Rep_formula_top[simp]: "Rep_formula ⊤ = UNIV"
unfolding top_formula_def
by (simp add: Abs_formula_inverse)

lemma Rep_formula_bot[simp]: "Rep_formula ⊥ = {}"
unfolding bot_formula_def
by (simp add: Abs_formula_inverse)

lemma Rep_formula_compl: "Rep_formula (- x) = - Rep_formula x"
unfolding uminus_formula_def
by (simp add: Abs_formula_inverse Rep_formula)

lemma Rep_formula_diff:
  "Rep_formula (x - y) = Rep_formula x - Rep_formula y"
unfolding minus_formula_def
by (simp add: Abs_formula_inverse Rep_formula)

lemmas eq_formula_iff = Rep_formula_inject [symmetric]

lemmas Rep_formula_boolean_algebra_simps =
  less_eq_formula_def less_formula_def eq_formula_iff
  Rep_formula_sup Rep_formula_inf Rep_formula_top Rep_formula_bot
  Rep_formula_compl Rep_formula_diff

instance proof
qed (unfold Rep_formula_boolean_algebra_simps, auto)
end
```

A.4.1.2 Tempo properties

In this section we define the tempo properties.

Tempo1: disjoint split

```
definition dlist_tempo1 :: "('a dlist ⇒ bool) ⇒ bool"
where
"dlist_tempo1 S ≡ ∀i j l. i ≤ j ⟶ ¬ ((S (l†..i) ∧ S (l†j..)))"
```

Tempo2: belonging iff

```
definition dlist_tempo2 :: "('a dlist ⇒ bool) ⇒ bool"
where
"dlist_tempo2 S ≡ ∀i l. S l ⟷ (S (l†..i) ∨ S (l†i..))"

definition dlist_tempo3 :: "('a dlist ⇒ bool) ⇒ bool"
where
"dlist_tempo3 S ≡ ∀i j l. j < i ⟶ (S (l†j..i) ⟷
  (S (l†..i) ∧ S (l†j..)))"

definition dlist_tempo4 :: "('a dlist ⇒ bool) ⇒ bool"
where
"dlist_tempo4 S ≡ ∀ l. S l ⟷ (∃i. S (l†i..(Suc i)))"

definition dlist_tempo5 :: "('a dlist ⇒ bool) ⇒ bool"
where
"dlist_tempo5 S ≡
  ∀ i j l. (i ≠ j ∧ i < (#l) ∧ j < (#l)) ⟶
    ¬(S (l†i..(Suc i)) ∧ S (l†j..(Suc j)))"

definition dlist_tempo6 :: "('a dlist ⇒ bool) ⇒ bool"
where
"dlist_tempo6 S ≡ ∀l. (∀ i j. ¬ S (l†i..j)) ⟷ ¬ S l"

definition dlist_tempo7 :: "('a dlist ⇒ bool) ⇒ bool"
where
"dlist_tempo7 S ≡ ∀l. (∃ i j. i < j ∧ S (l†i..j)) ⟷ S l"

definition dlist_tempo :: "('a dlist ⇒ bool) ⇒ bool"
where
"dlist_tempo S ≡ dlist_tempo1 S ∧ dlist_tempo2 S ∧
  dlist_tempo3 S ∧ dlist_tempo5 S ∧ dlist_tempo4 S ∧ dlist_tempo6 S ∧
  dlist_tempo7 S"

lemmas tempo_defs = dlist_tempo_def dlist_tempo1_def dlist_tempo2_def
  dlist_tempo3_def dlist_tempo5_def dlist_tempo4_def dlist_tempo6_def
  dlist_tempo7_def
```

```
lemma dlist_tempo_1_no_gap:
  "dlist_tempo1 S ⟹ ∀i l. ¬ ((S (l†..i) ∧ S (l†i..)))"
unfolding dlist_tempo1_def
by auto

corollary dlist_tempo_1_no_gap_append:
  "dlist_tempo1 S ⟹
    ∀zs xs ys. list_of_dlist zs = list_of_dlist xs @ list_of_dlist ys ⟶
    ¬ ((S xs ∧ S ys))"
using dlist_tempo_1_no_gap
by (metis Dlist_list_of_dlist append_eq_conv_conj slice_left_drop
  take_slice_right)
```

A.4.1.3 Tempo properties for list member

We use the naming convention of variable, but in fact, a variable is equivalent to a list membership: `var a = {xs | a ∈ list_of_dlist xs}`.

```
lemma dlist_tempo1_member: "dlist_tempo1 (λxs. Dlist.member xs a)"
unfolding dlist_tempo1_def Dlist.member_def List.member_def
by (meson distinct_in_set_slice1_not_in_slice2)

lemma dlist_tempo2_member: "dlist_tempo2 (λxs. Dlist.member xs a)"
unfolding dlist_tempo2_def Dlist.member_def List.member_def
by (metis (no_types, lifting) Un_iff set_slice )

lemma dlist_tempo3_member: "dlist_tempo3 (λxs. Dlist.member xs a)"
unfolding dlist_tempo3_def Dlist.member_def List.member_def
by (metis DiffD2 Un_iff distinct_slice_diff2 dlist_append_extreme_left
  dlist_append_extreme_right less_imp_le_nat set_append)

lemma dlist_tempo5_member: "dlist_tempo5 (λxs. Dlist.member xs a)"
unfolding dlist_tempo5_def Dlist.member_def List.member_def
by (metis Dlist_list_of_dlist Suc_leI disjoint_dlist_def disjoint_slice_suc
  distinct_list_of_dlist dlist_empty_slice dlist_member_suc_nth1 empty_slice
  less_Suc_eq_0_disj not_less_eq slice_singleton)

lemma dlist_tempo4_member: "dlist_tempo4 (λxs. Dlist.member xs a)"
unfolding dlist_tempo4_def Dlist.member_def List.member_def

by (metis dlist_member_suc_nth in_set_conv_nth in_set_dropD in_set_takeD
  list_of_dlist_Dlist set_remdups size_dlist_def slice_dlist_def)
```

```
lemma dlist_tempo6_member: "dlist_tempo6 (λxs. Dlist.member xs a)"
unfolding dlist_tempo6_def Dlist.member_def List.member_def
by (metis append_Nil in_set_conv_decomp in_set_conv_nth in_set_dropD
  in_set_takeD length_pos_if_in_set list_of_dlist_slice take_drop_suc)
```

```
lemma dlist_tempo7_member: "dlist_tempo7 (λxs. Dlist.member xs a)"
unfolding dlist_tempo7_def Dlist.member_def List.member_def
by (metis Un_iff dlist_append_extreme_left dlist_member_suc_nth2
  in_set_conv_nth lessI less_imp_le_nat set_append set_slice size_dlist_def)
```

```
theorem dlist_tempo_member: "dlist_tempo (λxs. Dlist.member xs a)"
unfolding dlist_tempo_def
by (simp add: dlist_tempo1_member dlist_tempo2_member dlist_tempo3_member
  dlist_tempo5_member dlist_tempo4_member dlist_tempo6_member
  dlist_tempo7_member)
```

A.4.1.4 Tempo properties for other operators

```
lemma dlist_tempo1_inf: "⟦dlist_tempo1 a; dlist_tempo1 b⟧ ⟹
  dlist_tempo1 (λzs. a zs ∧ b zs)"
unfolding dlist_tempo1_def
by simp
```

```
lemma dlist_tempo3_inf: "⟦dlist_tempo3 a; dlist_tempo3 b⟧ ⟹
  dlist_tempo3 (λzs. a zs ∧ b zs)"
unfolding dlist_tempo3_def
by auto
```

```
lemma dlist_tempo2_sup: "⟦dlist_tempo2 a; dlist_tempo2 b⟧ ⟹
  dlist_tempo2 (λzs. a zs ∨ b zs)"
unfolding dlist_tempo2_def
by auto
```

```
lemma dlist_tempo4_sup: "⟦dlist_tempo4 a; dlist_tempo4 b⟧ ⟹
  dlist_tempo4 (λzs. a zs ∨ b zs)"
unfolding dlist_tempo4_def
by blast
```

A.4.2 XBefore of distinct lists

```
definition dlist_xbefore :: "('a dlist ⇒ bool) ⇒ ('a dlist ⇒ bool) ⇒
  'a dlist ⇒ bool"
```

```
where
"dlist_xbefore a b xs ≡ ∃i. a (xs†..i) ∧ b (xs†i..)"
```

A.4.2.1 XBefore and temporal properties

```
lemma dlist_tempo1_xbefore: "⟦dlist_tempo1 a; dlist_tempo1 b⟧ ⟹
  dlist_tempo1 (dlist_xbefore a b)"
unfolding dlist_tempo1_def dlist_xbefore_def slice_slice_simps
by (smt le_add1 min.absorb2 min.cobounded1 slice_right_slice_left_absorb
  slice_right_slice_right_absorb)
```

A.4.2.2 XBefore and appending

```
lemma Rep_slice_append:
  "list_of_dlist zs = (list_of_dlist (zs†..i)) @ (list_of_dlist (zs†i..))"
by (metis distinct_append distinct_list_of_dlist distinct_slice_inter_empty
  list_of_dlist_Dlist remdups_id_iff_distinct slice_append)

lemma dlist_xbefore_append:
  "dlist_xbefore a b zs ⟷
  (∃xs ys. set (list_of_dlist xs) ∩ set (list_of_dlist ys) =
    {} ∧ a xs ∧ b ys ∧
    list_of_dlist zs = ((list_of_dlist xs) @ (list_of_dlist ys)))"
unfolding dlist_xbefore_def
by (metis Rep_slice_append append_Nil2 append_eq_conv_conj
  distinct_slice_inter_empty dlist_xbefore_def drop_take max_0L
  size_dlist_def slice_append slice_dlist_def slice_left_def slice_right_def
  take_slice_right)
```

A.4.2.3 XBefore, bot, top and idempotency

```
lemma dlist_xbefore_bot_1: "dlist_xbefore (λxs. False) b zs = False"
unfolding dlist_xbefore_def
by simp

corollary dlistset_xbefore_bot_1:
  "Collect (dlist_xbefore (λxs. False) b) = {}"
by (simp add: dlist_xbefore_bot_1)



lemma dlist_xbefore_bot_2: "dlist_xbefore a (λxs. False) zs = False"
unfolding dlist_xbefore_def
```

```
by simp

lemma dlistset_xbefore_bot_2:
  "Collect (dlist_xbefore a (λxs. False)) = {}"
by (simp add: dlist_xbefore_bot_2)

lemma dlist_xbefore_idem:
  "dlist_tempo1 a ⟹ dlist_xbefore a a zs = False"
unfolding dlist_xbefore_def dlist_tempo1_def
by blast

lemma dlistset_xbefore_idem:
  "dlist_tempo1 a ⟹ Collect (dlist_xbefore a a) = {}"
by (simp add: dlist_xbefore_idem)

lemma dlist_xbefore_implies_idem:
  "∀xs. b xs ⟶ a xs ⟹ dlist_tempo1 a ⟹ dlist_xbefore a b zs = False"
unfolding dlist_tempo1_def dlist_xbefore_def
by blast
```

A.4.2.4 XBefore neutral

```
lemma dlist_xbefore_neutral_1:
  "dlist_tempo1 a ⟹ dlist_xbefore (λxs. xs = dlist_of_list []) a zs = a zs"
by (metis (full_types) Dlist_list_of_dlist Rep_slice_append append.simps(1)
  dlist_of_list dlist_xbefore_def take_0 take_slice_right)

corollary dlistset_xbefore_neutral_1:
  "dlist_tempo1 a ⟹ Collect (dlist_xbefore (λxs. xs = Dlist []) a) = Collect a"
using dlist_xbefore_neutral_1 by auto

lemma dlist_xbefore_neutral_2:
  "dlist_tempo1 a ⟹ dlist_xbefore a (λxs. xs = Dlist []) zs = a zs"
by (smt Dlist_list_of_dlist append_Nil2 distinct_append distinct_list_of_dlist dlist_of_list

  dlist_xbefore_append list_of_dlist_empty)

corollary dlistset_xbefore_neutral_2:
  "dlist_tempo1 a ⟹ Collect (dlist_xbefore a (λxs. xs = Dlist [])) = Collect a"
using dlist_xbefore_neutral_2 by auto
```

A.4.2.5 XBefore associativity

```
theorem dlist_xbefore_assoc1:
  "dlist_tempo1 S ⟹ dlist_tempo1 T ⟹ dlist_tempo1 U ⟹
  (dlist_xbefore (dlist_xbefore S T) U zs) ⟷
    (dlist_xbefore S (dlist_xbefore T U) zs)"
unfolding dlist_xbefore_def slice_slice_simps dlist_tempo_def
apply auto
apply (metis diff_is_0_eq less_imp_le max_0L min_def not_le
  ordered_cancel_comm_monoid_diff_class.le_iff_add slice_dlist_def
  take_eq_Nil)
by (metis le_add1 min.absorb2)
```

```
corollary dlist_xbefore_assoc:
  "dlist_tempo1 S ⟹ dlist_tempo1 T ⟹ dlist_tempo1 U ⟹
  (dlist_xbefore (dlist_xbefore S T) U) =
    (dlist_xbefore S (dlist_xbefore T U))"
using dlist_xbefore_assoc1 by blast
```

```
corollary dlistset_xbefore_assoc:
  "dlist_tempo1 S ⟹ dlist_tempo1 T ⟹ dlist_tempo1 U ⟹
  Collect (dlist_xbefore (dlist_xbefore S T) U) =
    Collect (dlist_xbefore S (dlist_xbefore T U))"
by (simp add: dlist_xbefore_assoc)
```

A.4.2.6 XBefore equivalences

```
lemma dlist_tempo1_le_uniqueness:
  "dlist_tempo1 S ⟹ S (l†..i) ⟹ i ≤ j ⟹ ¬ S (l†j..)" and
  "dlist_tempo1 S ⟹ S (l†j..) ⟹ i ≤ j ⟹ ¬ S (l†..i)"
unfolding dlist_tempo1_def
by auto
```

```
lemma dlist_xbefore_not_sym:
  "dlist_tempo1 S ⟹ dlist_tempo1 T ⟹ dlist_xbefore S T xs ⟹
  dlist_xbefore T S xs ⟹ False"
by (metis dlist_xbefore_def le_cases dlist_tempo1_le_uniqueness)
```

```
corollary dlist_xbefore_and:
  "dlist_tempo1 S ⟹ dlist_tempo1 T ⟹
    ((dlist_xbefore S T zs) ∧ (dlist_xbefore T S zs)) = False"
using dlist_xbefore_not_sym by blast
```

```
corollary dlistset_xbefore_and:
  "dlist_tempo1 S ⟹ dlist_tempo1 T ⟹
  (Collect (dlist_xbefore S T)) ∩ (Collect (dlist_xbefore T S)) = {}"
using dlist_xbefore_and
by auto

lemma dlist_tempo2_left_absorb: "dlist_tempo2 S ⟹ S (l†i..) ⟹ S l"
unfolding dlist_tempo2_def
by auto

lemma dlist_tempo2_right_absorb: "dlist_tempo2 S ⟹ S (l†..i) ⟹ S l"
unfolding dlist_tempo2_def
by auto

lemma dlist_xbefore_implies_member1[simp]:
  "dlist_tempo2 S ⟹ dlist_xbefore S T l ⟹ S l"
by (meson dlist_xbefore_def dlist_tempo2_right_absorb)

lemma dlist_xbefore_implies_member2[simp]:
  "dlist_tempo2 T ⟹ dlist_xbefore S T l ⟹ T l"
by (meson dlist_xbefore_def dlist_tempo2_left_absorb)

lemma dlist_xbefore_or1:
  "dlist_tempo2 S ⟹ dlist_tempo2 T ⟹
  dlist_xbefore S T l ∨ dlist_xbefore T S l ⟹ S l ∧ T l"
using dlist_xbefore_implies_member1 dlist_xbefore_implies_member2 by blast


definition dlist_independent_events ::
  "('a dlist ⇒ bool) ⇒ ('a dlist ⇒ bool) ⇒ bool"
where
"dlist_independent_events S T ≡
  (∀i l. ¬ (S (l†i..(Suc i)) ∧ T (l†i..(Suc i))))"

lemma dlist_indepentent_events_member: "a ≠ b ⟹
  dlist_independent_events (λ dl. Dlist.member dl a) (λ dl. Dlist.member dl b)"
apply (simp add: dlist_independent_events_def Dlist.member_def List.member_def)
by (metis dlist_member_suc_nth1)
```

```
lemma dlist_and_split9:
  "dlist_independent_events S T ⟹
    dlist_tempo2 S ⟹ dlist_tempo2 T ⟹
    dlist_tempo3 S ⟹ dlist_tempo3 T ⟹
    dlist_tempo4 S ⟹ dlist_tempo4 T ⟹
  S l ∧ T l ⟷ (∃ i j. i ≤ j ∧
    ((S (l†..i) ∧ T (l†j..)) ∨ (S (l†j..) ∧ T (l†..i))))"
unfolding dlist_independent_events_def
  dlist_tempo2_def dlist_tempo3_def dlist_tempo4_def
by (metis le_refl not_less not_less_eq_eq)

lemma dlist_tempo_equiv_xor:
  "dlist_tempo1 S ⟹ dlist_tempo2 S ⟹
  ∀ l. S l ⟷ (∀ i. (S (l†..i) ∧ ¬ S (l†i..)) ∨ (¬ S (l†..i) ∧ S (l†i..)))"
unfolding tempo_defs
by (meson order_refl)

corollary dlist_tempo_equiv_not_eq: "dlist_tempo1 S ⟹ dlist_tempo2 S ⟹
  ∀ l. S l ⟷ (∀ i. S (l†..i) ≠ S (l†i..))"
using dlist_tempo_equiv_xor
by auto

lemma dlists_xbefore_or2:
  "dlist_independent_events S T ⟹
  dlist_tempo1 S ⟹ dlist_tempo1 T ⟹
  dlist_tempo2 S ⟹ dlist_tempo2 T ⟹
  dlist_tempo3 S ⟹ dlist_tempo3 T ⟹
  dlist_tempo4 S ⟹ dlist_tempo4 T ⟹
  S l ∧ T l ⟹ dlist_xbefore S T l ∨ dlist_xbefore T S l"
unfolding dlist_xbefore_def dlist_tempo_def
by (metis dlist_and_split9 dlist_tempo_equiv_not_eq
  dlist_tempo1_le_uniqueness)

theorem dlist_xbefore_or_one_list:
  "dlist_independent_events S T ⟹
  dlist_tempo1 S ⟹ dlist_tempo1 T ⟹
  dlist_tempo2 S ⟹ dlist_tempo2 T ⟹
  dlist_tempo3 S ⟹ dlist_tempo3 T ⟹
  dlist_tempo4 S ⟹ dlist_tempo4 T ⟹
  dlist_xbefore S T l ∨ dlist_xbefore T S l ⟷ S l ∧ T l"
using dlist_xbefore_or1 dlists_xbefore_or2 dlist_tempo_def
by blast

corollary dlist_xbefore_or:
  "dlist_independent_events S T ⟹
  dlist_tempo1 S ⟹ dlist_tempo1 T ⟹
  dlist_tempo2 S ⟹ dlist_tempo2 T ⟹
  dlist_tempo3 S ⟹ dlist_tempo3 T ⟹
  dlist_tempo4 S ⟹ dlist_tempo4 T ⟹
  (λzs. (dlist_xbefore S T zs) ∨ (dlist_xbefore T S zs)) =
    (λzs. S zs ∧ T zs)"
using dlist_xbefore_or_one_list
by blast

corollary dlistset_xbefore_or:
  "dlist_independent_events S T ⟹
  dlist_tempo1 S ⟹ dlist_tempo1 T ⟹
  dlist_tempo2 S ⟹ dlist_tempo2 T ⟹
  dlist_tempo3 S ⟹ dlist_tempo3 T ⟹
  dlist_tempo4 S ⟹ dlist_tempo4 T ⟹
  (Collect (dlist_xbefore S T)) ∪ (Collect (dlist_xbefore T S)) =
    Collect S ∩ Collect T"
using dlist_xbefore_or
by (smt Collect_cong Collect_conj_eq Collect_disj_eq)
```

A.4.2.7 XBefore transitivity

```
theorem dlist_xbefore_trans: "
  ⟦dlist_tempo1 a; dlist_tempo1 b; dlist_tempo1 c⟧ ⟹
  ⟦dlist_tempo2 a; dlist_tempo2 b; dlist_tempo2 c⟧ ⟹
  dlist_xbefore a b zs ∧ dlist_xbefore b c zs ⟹
  dlist_xbefore a c zs"
using dlist_xbefore_not_sym
by (metis dlist_tempo2_def dlist_xbefore_def)

corollary dlistset_xbefore_trans: "
  ⟦dlist_tempo1 a; dlist_tempo1 b; dlist_tempo1 c⟧ ⟹
  ⟦dlist_tempo2 a; dlist_tempo2 b; dlist_tempo2 c⟧ ⟹
  (Collect (dlist_xbefore a b) ∩ Collect (dlist_xbefore b c)) ⊆
    Collect (dlist_xbefore a c)"
using dlist_xbefore_trans
```

```
by auto
```

A.4.2.8 Boolean operators mixed with XBefore

```
theorem mixed_dlist_xbefore_or1: "
  dlist_xbefore (λxs. a xs ∨ b xs) c zs =
  ((dlist_xbefore a c zs) ∨ (dlist_xbefore b c zs))"
unfolding dlist_xbefore_def by auto

corollary mixed_dlistset_xbefore_or1: "
  Collect (dlist_xbefore (λxs. a xs ∨ b xs) c) =
  Collect (dlist_xbefore a c) ∪ Collect (dlist_xbefore b c)"
proof-
  have "Collect (λzs. (dlist_xbefore a c zs) ∨ (dlist_xbefore b c zs)) =
    (Collect (dlist_xbefore a c) ∪ Collect (dlist_xbefore b c))"
    by (simp add: Collect_disj_eq)
  thus ?thesis using mixed_dlist_xbefore_or1 by blast
qed

theorem mixed_dlist_xbefore_or2: "
  dlist_xbefore a (λxs. b xs ∨ c xs) zs =
  ((dlist_xbefore a b zs) ∨ (dlist_xbefore a c zs))"
unfolding dlist_xbefore_def by auto

corollary mixed_dlistset_xbefore_or2: "
  Collect (dlist_xbefore a (λxs. b xs ∨ c xs)) =
  Collect (dlist_xbefore a b) ∪ Collect (dlist_xbefore a c)"
proof-
  have "Collect (λzs. (dlist_xbefore a b zs) ∨ (dlist_xbefore a c zs)) =
    Collect (dlist_xbefore a b) ∪ Collect (dlist_xbefore a c)"
    by (simp add: Collect_disj_eq)
  thus ?thesis using mixed_dlist_xbefore_or2 by blast
qed

lemma and_dlist_xbefore_equiv_or_dlist_xbefore:
  "dlist_tempo2 a ⟹
  (a zs ∧ dlist_xbefore b c zs) ⟷
    (dlist_xbefore (λ xs. a xs ∧ b xs) c zs ∨
      dlist_xbefore b (λxs. a xs ∧ c xs) zs)"
proof-
  assume "dlist_tempo2 a"
  hence 0: "∀i xs. a xs ⟷ (a (xs†..i) ∨ a (xs†i..))"
```

```
    using dlist_tempo2_def by auto
  have "a zs ∧ dlist_xbefore b c zs ⟷
    a zs ∧ (∃i. b (zs†..i) ∧ c (zs†i..))"
    by (auto simp add: dlist_xbefore_def)
  thus ?thesis using 0 by (auto simp add: dlist_xbefore_def)
qed

corollary and_dlistset_xbefore_equiv_or_dlistset_xbefore:
  "dlist_tempo2 a ⟹
  ((Collect a) ∩ (Collect (dlist_xbefore b c))=
    (Collect (dlist_xbefore (λ xs. a xs ∧ b xs) c) ∪
      Collect (dlist_xbefore b (λxs. a xs ∧ c xs)))"
by (smt Collect_cong Collect_conj_eq Collect_disj_eq dlist_tempo2_def
  dlist_xbefore_def)

lemma dlist_xbefore_implies_not_sym_dlist_xbefore: "
  ⟦dlist_tempo1 a; dlist_tempo1 b⟧ ⟹
  dlist_xbefore a b zs ⟹ ¬ dlist_xbefore b a zs"
unfolding dlist_xbefore_def dlist_tempo1_def
by (meson nat_le_linear)

corollary dlistset_xbefore_implies_not_sym_dlistset_xbefore:
  "⟦dlist_tempo1 a; dlist_tempo1 b⟧ ⟹
  Collect (dlist_xbefore a b) ⊆ - Collect (dlist_xbefore b a)"
using dlist_xbefore_implies_not_sym_dlist_xbefore
by (metis (mono_tags, lifting) CollectD ComplI subsetI)

theorem mixed_not_dlist_xbefore: "dlist_independent_events a b ⟹
  ⟦dlist_tempo1 a; dlist_tempo1 b⟧ ⟹
  ⟦dlist_tempo2 a; dlist_tempo2 b⟧ ⟹
  ⟦dlist_tempo3 a; dlist_tempo3 b⟧ ⟹
  ⟦dlist_tempo4 a; dlist_tempo4 b⟧ ⟹
  (¬ (dlist_xbefore a b zs)) =
  ((¬ a zs) ∨ (¬ b zs) ∨ (dlist_xbefore b a zs))"
using dlist_xbefore_implies_not_sym_dlist_xbefore dlist_xbefore_or_one_list
by blast

corollary mixed_not_dlistset_xbefore: "dlist_independent_events a b ⟹
  ⟦dlist_tempo1 a; dlist_tempo1 b⟧ ⟹
  ⟦dlist_tempo2 a; dlist_tempo2 b⟧ ⟹
  ⟦dlist_tempo3 a; dlist_tempo3 b⟧ ⟹
  ⟦dlist_tempo4 a; dlist_tempo4 b⟧ ⟹
  (- Collect (dlist_xbefore a b)) =
  ((- Collect a) ∪ (- Collect b) ∪ Collect (dlist_xbefore b a))"
proof-
  assume 0: "dlist_independent_events a b" "dlist_tempo1 a" "dlist_tempo1 b"
  "dlist_tempo2 a" "dlist_tempo2 b" "dlist_tempo3 a" "dlist_tempo3 b"
  "dlist_tempo4 a" "dlist_tempo4 b"
  have "((- Collect a) ∪ (- Collect b) ∪ Collect (dlist_xbefore b a)) =
    ((Collect (λzs. ¬ a zs ∨ ¬ b zs)) ∪ Collect (dlist_xbefore b a))"
    by blast
  also have "... = (Collect (λzs. ¬ a zs ∨ ¬ b zs ∨ dlist_xbefore b a zs))"
    by blast
  hence "Collect (λzs. (¬ a zs) ∨ (¬ b zs) ∨ (dlist_xbefore b a zs)) =
    ((- Collect a) ∪ (- Collect b) ∪ Collect (dlist_xbefore b a))"
    "Collect (λzs. ¬ (dlist_xbefore a b zs)) =
      - Collect (dlist_xbefore a b)"
    by blast+
  thus ?thesis using 0 mixed_not_dlist_xbefore by blast
qed

theorem not_1_dlist_xbefore:
  "⟦ dlist_tempo1 a;  dlist_tempo2 b⟧ ⟹
    dlist_xbefore (λxs. ¬ a xs) b zs = b zs"
by (metis Dlist_list_of_dlist dlist_tempo_1_no_gap dlist_xbefore_def dlist_xbefore_implie
drop_0 slice_left_drop slice_right_take take_0)

corollary not_1_dlistset_xbefore:
  "⟦ dlist_tempo1 a;  dlist_tempo2 b⟧ ⟹
    Collect (dlist_xbefore (λxs. ¬ a xs) b) = Collect b"
using not_1_dlist_xbefore by blast

theorem not_2_dlist_xbefore:
  "⟦ dlist_tempo1 b; dlist_tempo2 a ⟧ ⟹ dlist_xbefore a (λxs. ¬ b xs) zs = a zs"
by (metis Dlist.empty_def append_Nil2 dlist_tempo_1_no_gap
  dlist_xbefore_append dlist_xbefore_implies_member1 drop_0 inf.commute
  inf_bot_left list.set(1) list_of_dlist_empty slice_left_drop
  slice_right_take take_0)

corollary not_2_dlistset_xbefore:
  "⟦ dlist_tempo1 b; dlist_tempo2 a ⟧ ⟹
  Collect (dlist_xbefore a (λxs. ¬ b xs)) = Collect a"
using not_2_dlist_xbefore by blast
```

```
lemma empty_dlist_implies_false[simp]:
  "⟦ dlist_tempo1 a; dlist_tempo2 a ⟧ ⟹ a (Dlist []) ⟹ False"
unfolding dlist_tempo1_def dlist_tempo2_def dlist_tempo3_def dlist_tempo4_def
  slice_left_def slice_right_def size_dlist_def slice_dlist_def
by (metis Dlist.empty_def list.size(3) list_of_dlist_empty nat_le_linear)


lemma dlist_inf_xbefore_trans:
  "⟦ dlist_tempo1 b; dlist_tempo3 b ⟧ ⟹ ((dlist_xbefore a b zs) ∧ (dlist_xbefore
b c zs)) ⟷
    (dlist_xbefore (dlist_xbefore a b) c) zs"
proof-
  assume 0: "dlist_tempo1 b" "dlist_tempo3 b"
  hence 1: "∃ i. (∃ j. a (zs†..i) ∧ b (zs†i..) ∧ b (zs†..j) ∧ c (zs†j..) ⟷
      a (zs†..i) ∧ b (zs†i..j) ∧ c (zs†j..))"
    by (metis slice_left_def slice_right_def)
  have 2: "(∃ x y. a (zs†..x) ∧ b (zs†x..) ∧ b (zs†..y) ∧ c (zs†y..)) ⟷
      (∃ x y. a (zs†..x) ∧ b (zs†x..y) ∧ c (zs†y..))"
    using 0
    by (metis (no_types, hide_lams) diff_zero dlist_empty_slice dlist_tempo1_le_uniqueness
dlist_tempo3_def dlist_tempo_1_no_gap drop_0 list_of_dlist_empty list_of_dlist_simps(3)
max_0L not_le slice_left_drop slice_right_def take_0)
  have 3: "((∃i. a (zs†..i) ∧ b (zs†i..)) ∧ (∃j. b (zs†..j) ∧ c (zs†j..))) ⟷

    (∃ i j. a (zs†..i) ∧ b (zs†i..) ∧ b (zs†..j) ∧ c (zs†j..))"
    "(∃i. (∃j. a (zs†..min i j) ∧ b (zs†j..i)) ∧ c (zs†i..)) ⟷
      (∃i j. a (zs†..min i j) ∧ b (zs†j..i) ∧ c (zs†i..))"
    by auto
  have 4: "(∃ x y. a (zs†..min x y) ∧ b (zs†x..y) ∧ c (zs†y..)) ⟷
    (∃ x y. a (zs†..x) ∧ b (zs†x..y) ∧ c (zs†y..))"
    using 0
    by (metis (no_types, lifting) Dlist.empty_def append_Nil2 dlist_empty_slice
dlist_tempo_1_no_gap_append list_of_dlist_empty min.cobounded1 min_def)
  have "(∃i j. a (zs†..i) ∧ b (zs†i..j) ∧ c (zs†j..)) ⟷
    (∃i j. a (zs†..min i j) ∧ b (zs†i..j) ∧ c (zs†j..))"
    using 4 by simp
  hence "(∃i j. a (zs†..i) ∧ b (zs†i..) ∧ b (zs†..j) ∧ c (zs†j..)) ⟷
    (∃i j. a (zs†..min i j) ∧ b (zs†i..j) ∧ c (zs†j..))"
    using 0 2 by simp
  hence "((∃i. a (zs†..i) ∧ b (zs†i..)) ∧ (∃j. b (zs†..j) ∧ c (zs†j..))) ⟷

    (∃i j. a (zs†..min i j) ∧ b (zs†i..j) ∧ c (zs†j..))"
    using 0 3 by simp
  hence "((∃i. a (zs†..i) ∧ b (zs†i..)) ∧ (∃j. b (zs†..j) ∧ c (zs†j..))) ⟷

    (∃j. (∃i. a (zs†..min i j) ∧ b (zs†i..j)) ∧ c (zs†j..))"
    using 3 by auto
  hence "(dlist_xbefore a b zs ∧ dlist_xbefore b c zs) ⟷
    (∃j. (∃i. a (zs†..min i j) ∧ b (zs†i..j)) ∧ c (zs†j..))"
    using dlist_xbefore_def by auto
  hence "(dlist_xbefore a b zs ∧ dlist_xbefore b c zs) ⟷
    (∃j. (∃i. a ((zs†..j)†..i) ∧ b ((zs†..j)†i..)) ∧ c (zs†j..))"
    by (simp add: min.commute slice_right_slice_left_absorb slice_right_slice_right_absor
  thus ?thesis unfolding dlist_xbefore_def by simp
qed

lemma dlistset_inf_xbefore_trans:
  "⟦ dlist_tempo1 b; dlist_tempo3 b ⟧ ⟹ (Collect  (dlist_xbefore a b) ∩ Collect
(dlist_xbefore b c)) =
    Collect (dlist_xbefore (dlist_xbefore a b) c)"
using dlist_inf_xbefore_trans
using Collect_cong Collect_conj_eq by blast
```

A.4.3 Formulas as ATF

In the following we prove that a formula is a valid type instantation for all ATF classes.

A.4.3.1 Basic properties of ATF

```
instantiation formula :: (type) temporal_faults_algebra_basic
begin

definition
  "neutral = Abs_formula { dlist_of_list [] }"

definition
  "xbefore a b = Abs_formula { zs .
    dlist_xbefore (λxs. xs ∈ Rep_formula a) (λys. ys ∈ Rep_formula b) zs }"

definition
  "tempo1 a = dlist_tempo1 (λxs. xs ∈ Rep_formula a)"

lemma Rep_formula_xbefore_to_dlist_xbefore:
  "Rep_formula (xbefore a b) =
  Collect (dlist_xbefore (λx. x ∈ Rep_formula a) (λy. y ∈ Rep_formula b))"
unfolding dlist_xbefore_def xbefore_formula_def
by (simp add: Abs_formula_inverse)

lemma Rep_formula_xbefore_bot_1: "Rep_formula (xbefore bot a) =
  Rep_formula bot"
unfolding xbefore_formula_def
by (simp add: Abs_formula_inverse dlist_xbefore_bot_1)

lemma Rep_formula_xbefore_bot_2: "Rep_formula (xbefore a bot) =
  Rep_formula bot"
unfolding xbefore_formula_def
by (simp add: Abs_formula_inverse dlist_xbefore_bot_2)

lemma Rep_formula_xbefore_neutral_1: "tempo1 a ⟹ Rep_formula (xbefore neutral
a) = Rep_formula a"
unfolding xbefore_formula_def neutral_formula_def tempo1_formula_def
apply (simp add: Abs_formula_inverse)
using dlistset_xbefore_neutral_1 by auto

lemma Rep_formula_xbefore_neutral_2: "tempo1 a ⟹ Rep_formula (xbefore a neutral)
= Rep_formula a"
unfolding xbefore_formula_def neutral_formula_def tempo1_formula_def
apply (simp add: Abs_formula_inverse)
using dlistset_xbefore_neutral_2 by auto

lemma Rep_formula_xbefore_not_idempotent:
  "tempo1 a ⟹ Rep_formula (xbefore a a) = Rep_formula bot"
unfolding xbefore_formula_def tempo1_formula_def
by (simp add: Abs_formula_inverse dlist_xbefore_idem)

lemma Rep_formula_xbefore_not_sym:
  "⟦ tempo1 a; tempo1 b⟧ ⟹
    Rep_formula (xbefore a b) ⊆ Rep_formula (-xbefore b a)"
unfolding xbefore_formula_def tempo1_formula_def uminus_formula_def
by (simp add: Abs_formula_inverse
  dlistset_xbefore_implies_not_sym_dlistset_xbefore)

instance proof
  fix a::"'a formula"
  show "xbefore bot a = bot"
  unfolding eq_formula_iff Rep_formula_xbefore_bot_1 by auto
  next
  fix a::"’a formula"
  show "xbefore a bot = bot"
  unfolding eq_formula_iff Rep_formula_xbefore_bot_2 by auto
  next
  fix a::"’a formula"
  assume "tempo1 a"
  thus "xbefore neutral a = a"
  unfolding eq_formula_iff
  using Rep_formula_xbefore_neutral_1 by auto
  next
  fix a::"’a formula"
  assume "tempo1 a"
  thus "xbefore a neutral = a"
  unfolding eq_formula_iff
  using Rep_formula_xbefore_neutral_2 by auto
  next
  fix a::"’a formula"
  assume "tempo1 a"
  thus "xbefore a a = bot"
  unfolding eq_formula_iff
  using Rep_formula_xbefore_not_idempotent by auto
  next
  fix a::"’a formula" and b::"’a formula"
  assume "tempo1 a" "tempo1 b"
  thus "xbefore a b ≤ - xbefore b a"
  unfolding eq_formula_iff less_eq_formula_def
  using Rep_formula_xbefore_not_sym by simp
  fix a::"’a formula" and b::"’a formula"
  assume "tempo1 a" "tempo1 b"
  thus "tempo1 (inf a b)"
  unfolding tempo1_formula_def
  by (simp add: dlist_tempo1_inf Rep_formula_inf)
qed

end
```

A.4.3.2 Associativity of ATF

```
instantiation formula :: (type) temporal_faults_algebra_assoc
begin
```

```
instance proof
  fix a::"’a formula" and b::"’a formula" and c::"’a formula"
  assume "tempo1 a" "tempo1 b" "tempo1 c"
  thus "xbefore (xbefore a b) c = xbefore a (xbefore b c)"
  unfolding xbefore_formula_def tempo1_formula_def
  by (simp add: Abs_formula_inverse dlist_xbefore_assoc)
qed

end
```

A.4.3.3 Equivalences in ATF

```
instantiation formula :: (type) temporal_faults_algebra_equivs
begin

definition
  "independent_events a b =
    dlist_independent_events
      (λxs. xs ∈ Rep_formula a) (λxs. xs ∈ Rep_formula b)"

definition
  "tempo2 a = dlist_tempo2 (λxs. xs ∈ Rep_formula a)"

definition
  "tempo3 a = dlist_tempo3 (λxs. xs ∈ Rep_formula a)"

definition
  "tempo4 a = dlist_tempo4 (λxs. xs ∈ Rep_formula a)"

instance proof
  fix a::"’a formula" and b::"’a formula"
  assume "tempo1 a" "tempo1 b"
  thus "inf (xbefore a b) (xbefore b a) = bot"
  unfolding xbefore_formula_def tempo1_formula_def bot_formula_def
    inf_formula_def
  by (simp add: dlistset_xbefore_and Abs_formula_inverse)
  next
  fix a::"’a formula" and b::"’a formula"
  assume "independent_events a b" "tempo1 a" "tempo1 b" "tempo2 a" "tempo2 b"
    "tempo3 a" "tempo3 b" "tempo4 a" "tempo4 b"
  thus "sup (xbefore a b) (xbefore b a) = inf a b"
  unfolding xbefore_formula_def tempo1_formula_def tempo2_formula_def
    tempo3_formula_def tempo4_formula_def independent_events_formula_def
    sup_formula_def inf_formula_def
  by (simp add: dlistset_xbefore_or Abs_formula_inverse)
  next
  fix a::"'a formula" and b::"'a formula"
  assume "tempo2 a" "tempo2 b"
  thus "tempo2 (sup a b)"
  unfolding tempo2_formula_def
  by (simp add: dlist_tempo2_sup Rep_formula_sup)
  next
  fix a::"'a formula" and b::"'a formula"
  assume "tempo3 a" "tempo3 b"
  thus "tempo3 (inf a b)"
  unfolding tempo3_formula_def
  by (simp add: dlist_tempo3_inf Rep_formula_inf)
  next
  fix a::"'a formula" and b::"'a formula"
  assume "tempo4 a" "tempo4 b"
  thus "tempo4 (sup a b)"
  unfolding tempo4_formula_def
  by (simp add: dlist_tempo4_sup Rep_formula_sup)
qed

end
```

A.4.3.4 Transitivity in ATF

```
instantiation formula :: (type) temporal_faults_algebra_trans
begin
instance proof
  fix a::"'a formula" and b::"'a formula" and c::"'a formula"
  assume "tempo1 a" "tempo1 b" "tempo1 c" "tempo2 a" "tempo2 b" "tempo2 c"
  thus "inf (xbefore a b) (xbefore b c) ≤ xbefore a c"
  unfolding tempo1_formula_def tempo2_formula_def xbefore_formula_def
    less_eq_formula_def inf_formula_def
  by (simp add: dlistset_xbefore_trans Abs_formula_inverse)
  next
  fix a::"'a formula" and b::"'a formula" and c::"'a formula"
  assume "tempo1 b" "tempo3 b"
  thus "inf (xbefore a b) (xbefore b c) = xbefore (xbefore a b) c"
  unfolding xbefore_formula_def inf_formula_def tempo1_formula_def
```

```
    tempo3_formula_def
  by (simp add: Abs_formula_inverse dlistset_inf_xbefore_trans)
qed
end
```

A.4.3.5 Mixed operators in ATF

```
instantiation formula :: (type) temporal_faults_algebra_mixed_ops
begin
instance proof
  fix a::"'a formula" and b::"'a formula" and c::"'a formula"
  show "xbefore (sup a b) c = sup (xbefore a c) (xbefore b c)"
  unfolding xbefore_formula_def sup_formula_def
  by (simp add: mixed_dlistset_xbefore_or1 Abs_formula_inverse)
  next
  fix a::"'a formula" and b::"'a formula" and c::"'a formula"
  show "xbefore a (sup b c) = sup (xbefore a b) (xbefore a c)"
  unfolding xbefore_formula_def sup_formula_def
  by (simp add: mixed_dlistset_xbefore_or2 Abs_formula_inverse)
  next
  fix a::"'a formula" and b::"'a formula"
  assume "independent_events a b" "tempo1 a" "tempo1 b" "tempo2 a" "tempo2 b"
    "tempo3 a" "tempo3 b" "tempo4 a" "tempo4 b"
  thus "(- xbefore a b) = (sup (sup (- a) (- b)) (xbefore b a))"
  by (simp add: Abs_formula_inverse xbefore_formula_def uminus_formula_def
    sup_formula_def independent_events_formula_def tempo1_formula_def
    tempo2_formula_def tempo3_formula_def tempo4_formula_def
    mixed_not_dlistset_xbefore)
  next
  fix a::"'a formula" and b::"'a formula" and c::"'a formula"
  assume "tempo2 a"
  thus "inf a (xbefore b c) =
    sup (xbefore (inf a b) c) (xbefore b (inf a c))"
  apply (auto simp add: xbefore_formula_def sup_formula_def inf_formula_def
    tempo2_formula_def Abs_formula_inverse)
  using and_dlistset_xbefore_equiv_or_dlistset_xbefore Abs_formula_inverse
  by fastforce
  next
  fix a::"'a formula" and b::"'a formula"
  assume "tempo1 a" "tempo2 b"
  thus "xbefore (- a) b = b"
  unfolding xbefore_formula_def tempo1_formula_def tempo2_formula_def
```

```
    uminus_formula_def
  by (auto simp add: Abs_formula_inverse not_1_dlistset_xbefore
    Rep_formula_inverse)
  next
  fix a::"'a formula" and b::"'a formula"
  assume "tempo1 b" "tempo2 a"
  thus "xbefore a (- b) = a"
  unfolding xbefore_formula_def tempo1_formula_def tempo2_formula_def
    uminus_formula_def
  by (auto simp add: Abs_formula_inverse not_2_dlistset_xbefore
    Rep_formula_inverse)
qed
end
```

A.4.4 Equivalence of the new definition of XBefore with the old one

```
definition old_dlist_xbefore
where
"old_dlist_xbefore S T zs ≡
  (∃ xs ys. S xs ∧ ¬ T xs ∧ T ys ∧ ¬ S ys ∧
    set (list_of_dlist xs) ∩ set (list_of_dlist ys) = {} ∧
    list_of_dlist zs = (list_of_dlist xs) @ (list_of_dlist ys))"

theorem old_dlist_xbefore_equals_new_xbefore:
  "⟦ dlist_tempo1 S; dlist_tempo1 T ⟧ ⟹
    dlist_xbefore S T zs = old_dlist_xbefore S T zs"
unfolding dlist_xbefore_append old_dlist_xbefore_def
using dlist_tempo_1_no_gap_append
by blast
```

A.4.5 Soundness and completeness on the mapping rules

```
theorem temporal_faults_algebra_mapping_soundness:
    "∀ (f_1::'a formula) (f_2::'a formula). ∃ S. ((Rep_formula f_1 = S ∧ Rep_formula
f_2 = S) ⟷ f_1 = f_2)"
by blast

theorem temporal_faults_algebra_mapping_completeness:
    "∀ (S::'a dlist set). ∃ f::'a formula. Rep_formula f = S"
using Abs_formula_inverse by auto
```