



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-  
125 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 Dibia 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 Árvores 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 lógica 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). 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. 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.

<sup>1</sup> **Keywords:** Simulink, CSP, FDR, Fault Tree Analysis, Temporal Fault Trees, Dynamic Fault Trees, Isabelle/HOL, Pandora, Fault Injection





# List of figures

Figure 1 – Strategy overview . . . . .	30
Figure 2 – Relation of two events with duration . . . . .	38
Figure 3 – Static Fault Tree (SFT) symbols using a free commercial tool . . . . .	42
Figure 4 – SFT symbols as in the Fault Tree Handbook . . . . .	43
Figure 5 – SFT gates . . . . .	44
Figure 6 – Very simple example of a fault tree . . . . .	44
Figure 7 – TFT-specific gates . . . . .	45
Figure 8 – TFT small example . . . . .	46
Figure 9 – DFTs's original gates symbols . . . . .	48
Figure 10 – Dynamic Fault Trees's (DFTs's) [1, 2] gates symbols . . . . .	48
Figure 11 – DFT example . . . . .	51
Figure 12 – A diagram for a truth table . . . . .	53
Figure 13 – A BDD for the expression $A \vee (\neg B \wedge C)$ . . . . .	54
Figure 14 – DT for variables $X$ and $Y$ . . . . .	54
Figure 15 – DT for the formula $(X \wedge Y) \vee ((X < Y) \wedge Z)$ . . . . .	55
Figure 16 – ZBDD example of combination set $\{a, b\}$ . . . . .	56
Figure 17 – Non-coherent FT college student's example . . . . .	60
Figure 18 – Gas detection system . . . . .	60
Figure 19 – FT for a generic failure in the gas detection system . . . . .	61
Figure 20 – <i>Coherent</i> FT for the most critical outcome of the gas detection system	62
Figure 21 – <i>Non-coherent</i> FT for the most critical outcome of the gas detection system	62
Figure 22 – Block diagram of the ACS provided by EMBRAER (nominal model) .	63
Figure 23 – Internal diagram of the monitor component (Figure 22 (A)). . . . .	64
Figure 24 – Isabelle/HOL window, showing the basic symmetry theorem . . . . .	69
Figure 25 – Status of this thesis using the strategy overview (see Figure 1) . . . . .	90



# List of tables

Table 1	–	TTT of TFT's operators and sequence value numbers . . . . .	46
Table 2	–	TTT of a simple example . . . . .	47
Table 3	–	Dynamic Fault Tree (DFT) [1, 2] conversion to calculate probability of top-level event . . . . .	49
Table 4	–	Algebraic model of DFT gates with inputs $A$ and $B$ . . . . .	50
Table 5	–	Date-of-occurrence function for operators defined in [3] . . . . .	50
Table 6	–	Truth table for a formula outputs with three variables ( $A$ , $B$ , and $C$ ) . .	53
Table 7	–	Annotations table of the ACS provided by EMBRAER . . . . .	67
Table 8	–	Tasks schedule . . . . .	89



# List of abbreviations and acronyms

AADL	Architecture Analysis and Design Language p. <a href="#">27</a>
AL	Activation Logic pp. <a href="#">29–32</a> , <a href="#">80</a> , <a href="#">81</a> , <a href="#">89</a> , <a href="#">90</a>
AFP	archive of formal proofs p. <a href="#">68</a>
ATF	Algebra of Temporal Faults pp. <a href="#">24</a> , <a href="#">29</a> , <a href="#">31</a> , <a href="#">32</a> , <a href="#">39</a> , <a href="#">56</a> , <a href="#">66</a> , <a href="#">71–74</a> , <a href="#">77</a> , <a href="#">78</a> , <a href="#">80</a> , <a href="#">81</a> , <a href="#">85–87</a> , <a href="#">89–91</a> , <a href="#">103</a> , <a href="#">118–120</a>
BDD	Binary Decision Diagram pp. <a href="#">15</a> , <a href="#">25</a> , <a href="#">27</a> , <a href="#">28</a> , <a href="#">39</a> , <a href="#">43</a> , <a href="#">49</a> , <a href="#">52–58</a> , <a href="#">91</a>
BN	Bayesian network p. <a href="#">49</a>
CML	COMPASS Modelling Language p. <a href="#">36</a>
CPN	coloured Petri-net p. <a href="#">49</a>
CSP	Communicating Sequential Processes p. <a href="#">36</a>
CSP <sub>M</sub>	Communicating Sequential Processes pp. <a href="#">27</a> , <a href="#">31</a> , <a href="#">39</a> , <a href="#">64</a> , <a href="#">66</a> , <a href="#">71</a>
CTMC	continuous-time Markov chain pp. <a href="#">28</a> , <a href="#">49</a>
DBN	dynamic bayesian network p. <a href="#">28</a>
DD	Dependence Diagram p. <a href="#">37</a>
DFT	Dynamic Fault Tree pp. <a href="#">17</a> , <a href="#">26–29</a> , <a href="#">31</a> , <a href="#">33</a> , <a href="#">37</a> , <a href="#">39–41</a> , <a href="#">44</a> , <a href="#">46–52</a> , <a href="#">55</a> , <a href="#">56</a> , <a href="#">66</a> , <a href="#">71</a> , <a href="#">75</a> , <a href="#">91</a>
DNF	disjunctive normal form pp. <a href="#">40</a> , <a href="#">46</a> , <a href="#">49</a> , <a href="#">55</a> , <a href="#">57</a> , <a href="#">75</a> , <a href="#">89–91</a>
DRBD	Dynamic Reliability Block Diagram p. <a href="#">37</a>
DT	dependency tree pp. <a href="#">28</a> , <a href="#">47</a> , <a href="#">52</a> , <a href="#">54</a> , <a href="#">55</a>
DTMC	discrete-time Markov chain pp. <a href="#">28</a> , <a href="#">37</a> , <a href="#">47</a> , <a href="#">52</a> , <a href="#">91</a>
FBA	Free Boolean Algebra pp. <a href="#">25</a> , <a href="#">27</a> , <a href="#">29</a> , <a href="#">39</a> , <a href="#">51</a> , <a href="#">57</a> , <a href="#">68</a> , <a href="#">71–73</a> , <a href="#">86</a> , <a href="#">91</a>
FDR	Failures and Divergences Refinement pp. <a href="#">27</a> , <a href="#">64</a> , <a href="#">66</a>
FMEA	Failure Modes and Effects Analysis pp. <a href="#">29</a> , <a href="#">37</a>
FSM	Finite State Machine p. <a href="#">52</a>
FT	fault tree pp. <a href="#">15</a> , <a href="#">25–27</a> , <a href="#">29–34</a> , <a href="#">36</a> , <a href="#">39–42</a> , <a href="#">44</a> , <a href="#">46</a> , <a href="#">51</a> , <a href="#">52</a> , <a href="#">58–62</a> , <a href="#">66</a> , <a href="#">71</a> , <a href="#">77</a> , <a href="#">79</a> , <a href="#">81</a> , <a href="#">89</a>
FTA	Fault Tree Analysis pp. <a href="#">25</a> , <a href="#">27</a> , <a href="#">29</a> , <a href="#">30</a> , <a href="#">39–42</a> , <a href="#">59</a> , <a href="#">77</a>
HCAS	cardiac assist system p. <a href="#">50</a>
HiP-HOPS	Hierarchically Performed Hazard Origin and Propagation Studies pp. <a href="#">27–29</a> , <a href="#">34</a> , <a href="#">41</a> , <a href="#">66</a>
HLPN	high-level Petri-net p. <a href="#">52</a>
HOL	higher-order logic p. <a href="#">68</a>
Isar	Intelligible semi-automated reasoning pp. <a href="#">39</a> , <a href="#">68</a>

ITL	Interval Temporal Logic p. <a href="#">52</a>
LTL	linear temporal logic p. <a href="#">44</a>
MCS	minimal cut set pp. <a href="#">25</a> , <a href="#">40</a> , <a href="#">43</a> , <a href="#">46</a> , <a href="#">49</a>
MCSeq	minimal cut sequence pp. <a href="#">26</a> , <a href="#">31</a> , <a href="#">45</a> , <a href="#">46</a> , <a href="#">49</a> , <a href="#">52</a> , <a href="#">54–56</a> , <a href="#">77</a> , <a href="#">79</a> , <a href="#">80</a>
PN	Petri-net p. <a href="#">35</a>
ROBDD	Reduced Ordered Binary Decision Diagram pp. <a href="#">52–54</a>
SBDD	Sequential Binary Decision Diagram pp. <a href="#">28</a> , <a href="#">49</a> , <a href="#">52</a> , <a href="#">56</a>
SFT	Static Fault Tree pp. <a href="#">15</a> , <a href="#">26–28</a> , <a href="#">31</a> , <a href="#">33</a> , <a href="#">37</a> , <a href="#">39–44</a> , <a href="#">46</a> , <a href="#">49</a> , <a href="#">51</a> , <a href="#">52</a> , <a href="#">54</a> , <a href="#">55</a> , <a href="#">57</a> , <a href="#">58</a> , <a href="#">66</a> , <a href="#">71</a> , <a href="#">75</a> , <a href="#">91</a>
SoS	System of Systems pp. <a href="#">30</a> , <a href="#">34</a>
SWN	stochastic well-formed net p. <a href="#">49</a>
SysML	Systems Modelling Language pp. <a href="#">30</a> , <a href="#">36</a>
TFT	Temporal Fault Tree pp. <a href="#">26–29</a> , <a href="#">31</a> , <a href="#">33</a> , <a href="#">39–41</a> , <a href="#">44–46</a> , <a href="#">48</a> , <a href="#">49</a> , <a href="#">51</a> , <a href="#">52</a> , <a href="#">66</a> , <a href="#">71</a> , <a href="#">75</a>
TTT	Temporal Truth Table pp. <a href="#">17</a> , <a href="#">28</a> , <a href="#">45</a> , <a href="#">46</a> , <a href="#">54</a>
UML	Unified Modelling Language p. <a href="#">36</a>
Z	Z Notation pp. <a href="#">52</a> , <a href="#">68</a>
ZBDD	Zero-suppressed Binary Decision Diagram pp. <a href="#">52</a> , <a href="#">55</a>

# Fault tree gates

AND	$\wedge$ . Used in <a href="#">SFT</a> , <a href="#">TFT</a> , and <a href="#">DFT</a> . pp. <a href="#">25</a> , <a href="#">26</a> , <a href="#">39</a> , <a href="#">42</a> , <a href="#">44</a> , <a href="#">46</a> , <a href="#">49</a> , <a href="#">50</a> , <a href="#">56</a> , <a href="#">58</a> , <a href="#">59</a> , <a href="#">66</a> , <a href="#">67</a> , <a href="#">75</a> , <a href="#">78</a> , <a href="#">86</a> , <a href="#">91</a>
CSp	cold spare. Used in <a href="#">DFT</a> . pp. <a href="#">26</a> , <a href="#">40</a> , <a href="#">48</a> , <a href="#">50</a> , <a href="#">52</a> , <a href="#">56</a>
FDEP	functional dependency. Used in <a href="#">DFT</a> . pp. <a href="#">26</a> , <a href="#">40</a> , <a href="#">48</a> , <a href="#">50</a>
IBefore	inclusive-before. Used in structure expressions of <a href="#">DFT</a> . pp. <a href="#">49</a> , <a href="#">50</a> , <a href="#">56</a>
NIBefore	non-inclusive-before. Used in structure expressions of <a href="#">DFT</a> . pp. <a href="#">49</a> , <a href="#">50</a>
NOT	$\neg$ . Used in non-coherent trees. pp. <a href="#">26</a> , <a href="#">28</a> , <a href="#">31</a> , <a href="#">39</a> , <a href="#">44</a> , <a href="#">58</a> , <a href="#">59</a> , <a href="#">75</a> , <a href="#">78</a> , <a href="#">79</a> , <a href="#">89</a> , <a href="#">91</a>
OR	$\vee$ . Used in <a href="#">SFT</a> , <a href="#">TFT</a> , and <a href="#">DFT</a> . pp. <a href="#">25</a> , <a href="#">26</a> , <a href="#">39</a> , <a href="#">42</a> , <a href="#">44</a> , <a href="#">46</a> , <a href="#">49</a> , <a href="#">50</a> , <a href="#">58</a> , <a href="#">66</a> , <a href="#">79</a> , <a href="#">86</a> , <a href="#">87</a> , <a href="#">91</a>
PAND	priority-AND. Used in <a href="#">SFT</a> , <a href="#">TFT</a> , and <a href="#">DFT</a> . pp. <a href="#">26</a> , <a href="#">39</a> , <a href="#">40</a> , <a href="#">44–46</a> , <a href="#">48</a> , <a href="#">50</a> , <a href="#">52</a> , <a href="#">56</a> , <a href="#">77</a>
POR	priority-OR. Used in <a href="#">TFT</a> . pp. <a href="#">44</a> , <a href="#">46</a> , <a href="#">49</a>
SAND	simultaneous-AND. Used in <a href="#">TFT</a> . pp. <a href="#">44</a> , <a href="#">46</a> , <a href="#">48</a> , <a href="#">49</a>
SEQ	sequence enforcing. Used in <a href="#">DFT</a> . pp. <a href="#">26</a> , <a href="#">40</a> , <a href="#">48</a> , <a href="#">50</a>
SIMLT	simultaneous. Used in structure expressions of <a href="#">DFT</a> . pp. <a href="#">49</a> , <a href="#">50</a>
WSp	warm spare. Used in <a href="#">DFT</a> . pp. <a href="#">26</a> , <a href="#">56</a>
XBefore	exclusive-before. Proposed in this work. pp. <a href="#">24</a> , <a href="#">71–76</a> , <a href="#">85–87</a> , <a href="#">89</a> , <a href="#">91</a> , <a href="#">118</a> , <a href="#">119</a>





# Contents

<b>1</b>	<b>INTRODUCTION</b>	<b>25</b>
<b>1.1</b>	<b>Research questions</b>	<b>27</b>
<b>1.2</b>	<b>Proposed solution</b>	<b>29</b>
<b>1.3</b>	<b>Contributions</b>	<b>32</b>
<b>1.4</b>	<b>Thesis organization</b>	<b>32</b>
<b>2</b>	<b>BASIC CONCEPTS</b>	<b>33</b>
<b>2.1</b>	<b>Systems, dependability, and fault modelling</b>	<b>33</b>
2.1.1	Systems	33
2.1.2	Dependability	34
2.1.3	Fault Modelling	36
<b>2.2</b>	<b>Time relation of fault events</b>	<b>37</b>
<b>3</b>	<b>ANALYSIS AND TOOLS</b>	<b>39</b>
<b>3.1</b>	<b>Fault Tree Analysis and structure expressions</b>	<b>39</b>
3.1.1	Static Fault Trees	41
3.1.2	Temporal Fault Trees	44
3.1.3	Dynamic Fault Trees	47
<b>3.2</b>	<b>Structure expression analysis</b>	<b>51</b>
3.2.1	Stateful methods and temporal logic analysis	52
3.2.2	Binary Decision Diagrams	52
3.2.3	Dependency tree	54
3.2.4	Zero-suppressed Binary Decision Diagrams	55
3.2.5	Sequential Binary Decision Diagrams	56
<b>3.3</b>	<b>Free Boolean Algebras</b>	<b>57</b>
<b>3.4</b>	<b>Using the NOT operator in static fault trees</b>	<b>58</b>
3.4.1	Non-coherent fault tree misleads	59
3.4.2	Usefulness of NOT gates in FTA	59
<b>3.5</b>	<b>Systems nominal model and fault injection</b>	<b>63</b>
<b>3.6</b>	<b>Isabelle/HOL</b>	<b>68</b>
<b>4</b>	<b>A FREE ALGEBRA TO EXPRESS STRUCTURE EXPRESSIONS OF ORDERED EVENTS</b>	<b>71</b>
<b>4.1</b>	<b>Temporal properties (<i>tempo</i>)</b>	<b>73</b>
<b>4.2</b>	<b>XBefore laws</b>	<b>74</b>
<b>4.3</b>	<b>Qualitative and quantitative properties</b>	<b>77</b>

4.3.1	Formal acceptance criteria . . . . .	79
<b>4.4</b>	<b>Propositions . . . . .</b>	<b>80</b>
4.4.1	Soundness and completeness of ATF . . . . .	80
4.4.2	AL concepts . . . . .	80
<b>5</b>	<b>REASONING FAULT ACTIVATION . . . . .</b>	<b>83</b>
<b>6</b>	<b>CASE STUDY . . . . .</b>	<b>85</b>
6.1	Structure expressions with Boolean operators . . . . .	85
6.2	Structure expressions with XBefore . . . . .	86
<b>7</b>	<b>CONCLUSION . . . . .</b>	<b>89</b>
7.1	Status . . . . .	89
7.2	Next steps in this thesis . . . . .	90
7.3	Future work, out of the scope of this thesis . . . . .	91
	<b>BIBLIOGRAPHY . . . . .</b>	<b>93</b>
	<b>APPENDIX . . . . .</b>	<b>101</b>
	<b>APPENDIX A – FORMAL PROOFS IN ISABELLE/HOL . . . . .</b>	<b>103</b>
<b>A.1</b>	<b>Sliceable . . . . .</b>	<b>103</b>
A.1.1	Disjoint elements and sliceable . . . . .	104
A.1.2	n-th element in a sliceable . . . . .	104
A.1.3	Theorems for sliceable . . . . .	104
<b>A.2</b>	<b>Sliceable distinct lists . . . . .</b>	<b>108</b>
A.2.1	Properties of sliceable distinct lists . . . . .	111
<b>A.3</b>	<b>Algebra of Temporal Faults . . . . .</b>	<b>118</b>
A.3.1	Basic Algebra of Temporal Faults (ATF) operators and tempo1 . . . . .	118
A.3.2	Definition of associativity of exclusive-before (XBefore) . . . . .	118
A.3.3	Equivalences in the ATF and properties . . . . .	118
A.3.4	XBefore transitivity . . . . .	119
A.3.5	Mixed operators in ATF . . . . .	119
A.3.6	Theorems in the context of ATF . . . . .	120

# 1 Introduction

1 2

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 conjunctive (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

---

<sup>1</sup> AD Note: use all figure captions before figures (ABNT)

<sup>2</sup> AD Note: fix table lines (ABNT)

events.

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<sup>3</sup> 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.

**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.

<sup>3</sup> Pandora stands for: P-AND-ORA, which translates to Priority AND, Time.

Hierarchically Performed Hazard Origin and Propagation Studies<sup>4</sup> (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 (EMBRAER), 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<sup>5</sup> (CSP<sub>M</sub>) to allow an automatic analysis using the model checker FDR. Thus, our strategy required the translation from Simulink to CSP<sub>M</sub> [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, EMBRAER, we showed that our automatically created failure expressions match with the engineer's provided ones or are better (a weaker proposition).

## 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

<sup>4</sup> <<http://www.hip-hops.eu/>>

<sup>5</sup> This variant “M” is the machine-readable version of CSP.

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.

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

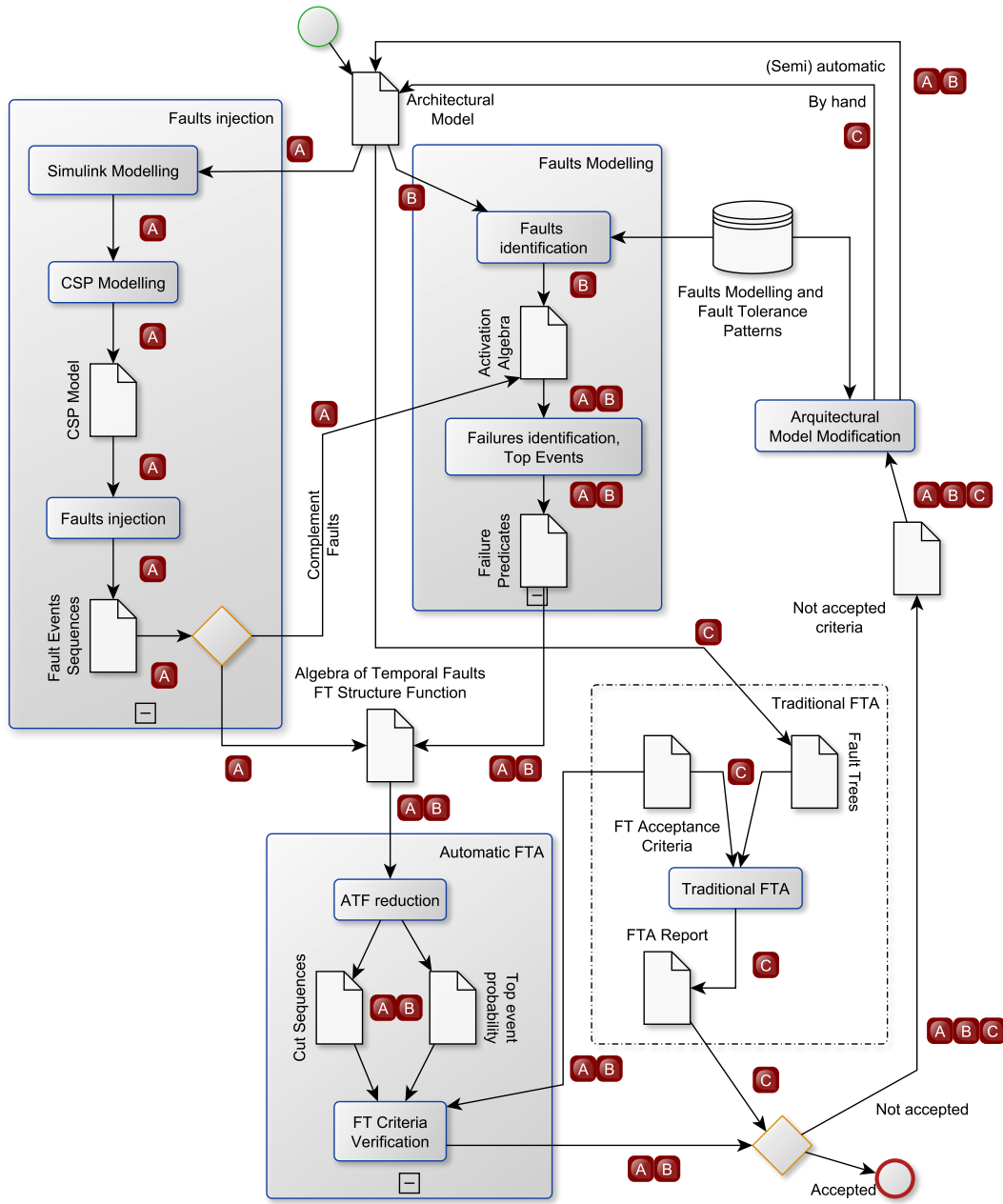
6

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<sup>7</sup> 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. In Figure 1 the strategy starts in the top (green) node and ends in the bottom (red) node. Fault events are either extracted from a nominal model with injected faults (Figure 1, path A), or modelled using a proposed notation, called Activation Logic (**AL**) (Figure 1, path B). We depict traditional **FTA** in path C to compare to our strategy and

<sup>6</sup> AD Note: Continue from here: what's denotational semantics?

<sup>7</sup> 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>.

because we still need part of the traditional FTA to obtain the acceptance criteria, which are the expected properties of system's FTs.



**Figure 1** – Strategy overview

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 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 **FT**s 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<sub>2</sub>**: (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.

The “Faults injection” block in Figure 1, path A, is obtained from part of our work reported in [28, 27]. It starts with Simulink modelling, converts the model to **CSP<sub>M</sub>** 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. As fault names are obtained directly from components and subsystems in a Simulink model, **AL** (in the “Fault Modelling” group) allows them to be modified or complemented. **AL** allows reasoning about faults that are not modelled in Simulink as, for example, common cause or environmental faults. Path A aims at answering the research question **RQ<sub>4</sub>**. Given the flexibility of the **AL** notation, it can also be used directly (path B) to model faults formally, reasoning about basic fault events and top-event failures, which are related to **RQ<sub>3</sub>**. 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, which is related to **RQ<sub>1</sub>**.

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<sub>5</sub>**.

### 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, EMBRAER (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:

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).

## 2 Basic concepts

Means to dependability<sup>1 2</sup> 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<sup>3</sup> 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.<sup>4 5</sup> The *total state* of a system are the means to implement its function and is defined as the set of the following states:

---

<sup>1</sup> AM Note: Está muito filosófico isto. É assim mesmo ou seja, não há uma definição para dependabilidade?

<sup>2</sup> AD Note: Eu explico dependabilidade a seguir.

<sup>3</sup> 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.

<sup>4</sup> AM Note: Você continua muito filosófico aqui. Quero ver para onde vamos (uso)...

<sup>5</sup> AD Note: Isso faz parte do contexto para quando mencionar os termos eles terem sido introduzidos.

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.

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 be 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<sup>6</sup> ([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

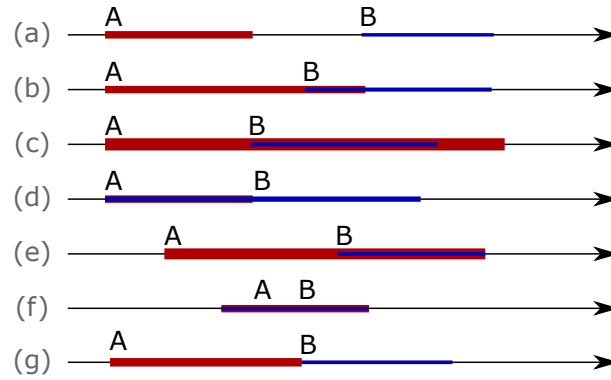
The most general case for time relations is to consider that each fault event has a continuous time duration. They are the basis on how fault events discretisation are defined. The point of view in this work is the analysis of the effects caused by a combination of faults in a snapshot of a system state. In Figure 2 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 2 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

<sup>6</sup> Also known as Reliability Block Diagram ([RBD](#)).

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]).



**Figure 2** – 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.

**FBA**s and **BDD**s 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 **BDD**s and derived techniques in Section 3.2, and **FBA**s 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<sub>M</sub>** 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**<sup>1</sup> gate was first defined for **SFTs**, but its analysis was left open in the handbook.

---

<sup>1</sup> AD Note: Eles são mencionados no FT handbook. Yannis viu a oportunidade de explorá-lo, assim

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 section.

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 3, or as a circle with the text of the basic event, as shown in Figure 4. Top-level and intermediary events are drawn as a rectangle (for the text) and a gate below it, as shown in Figures 3 and 4. 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 4 depicts traditional gates as specified in the Fault Tree Handbook, and Figure 3 shows an FT using the Fault Tree Analyser<sup>2</sup>—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 5.

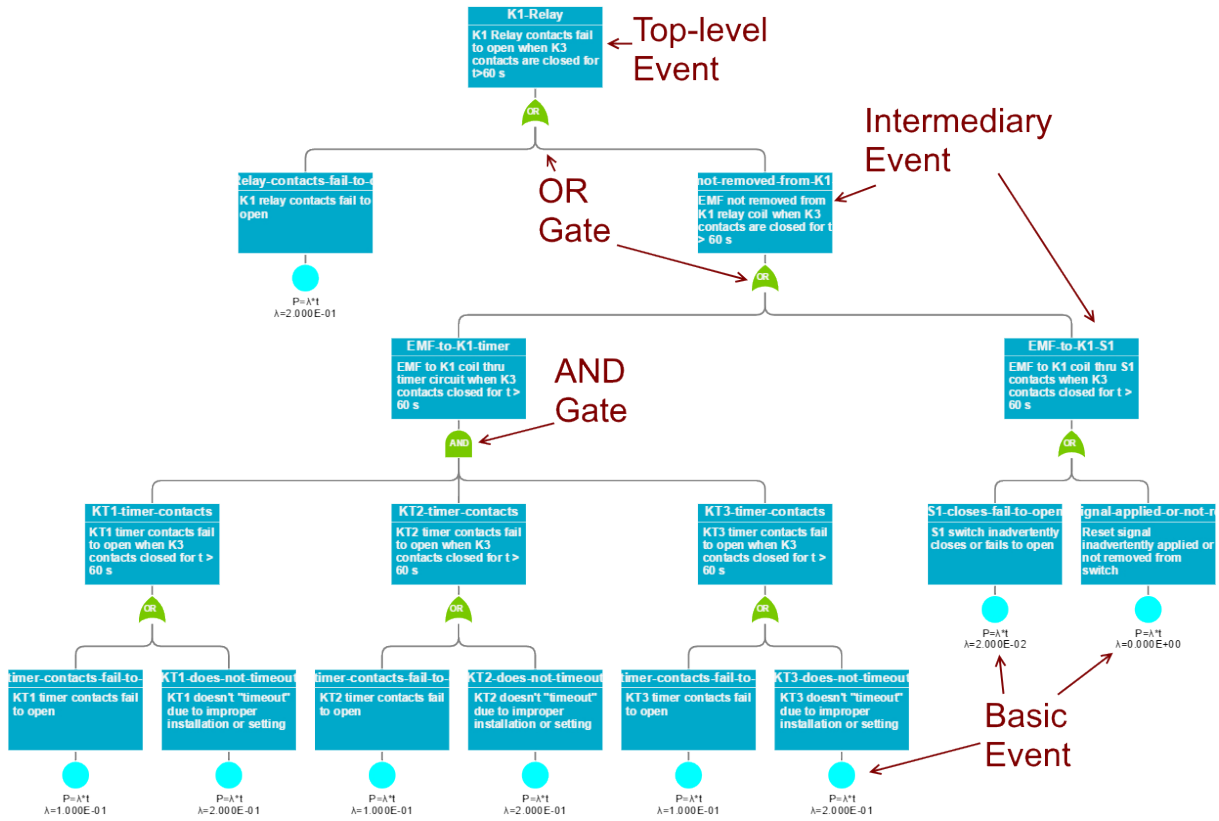


Figure 3 – SFT symbols using a free commercial tool

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 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:

- $\{a\}$ : only  $a$  occurs;
- $\{a, b\}$ :  $a$  and  $b$  occur in any order;
- $\{a, c\}$ :  $a$  and  $c$  occur in any order;
- $\{a, b, c\}$ : 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 6 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

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

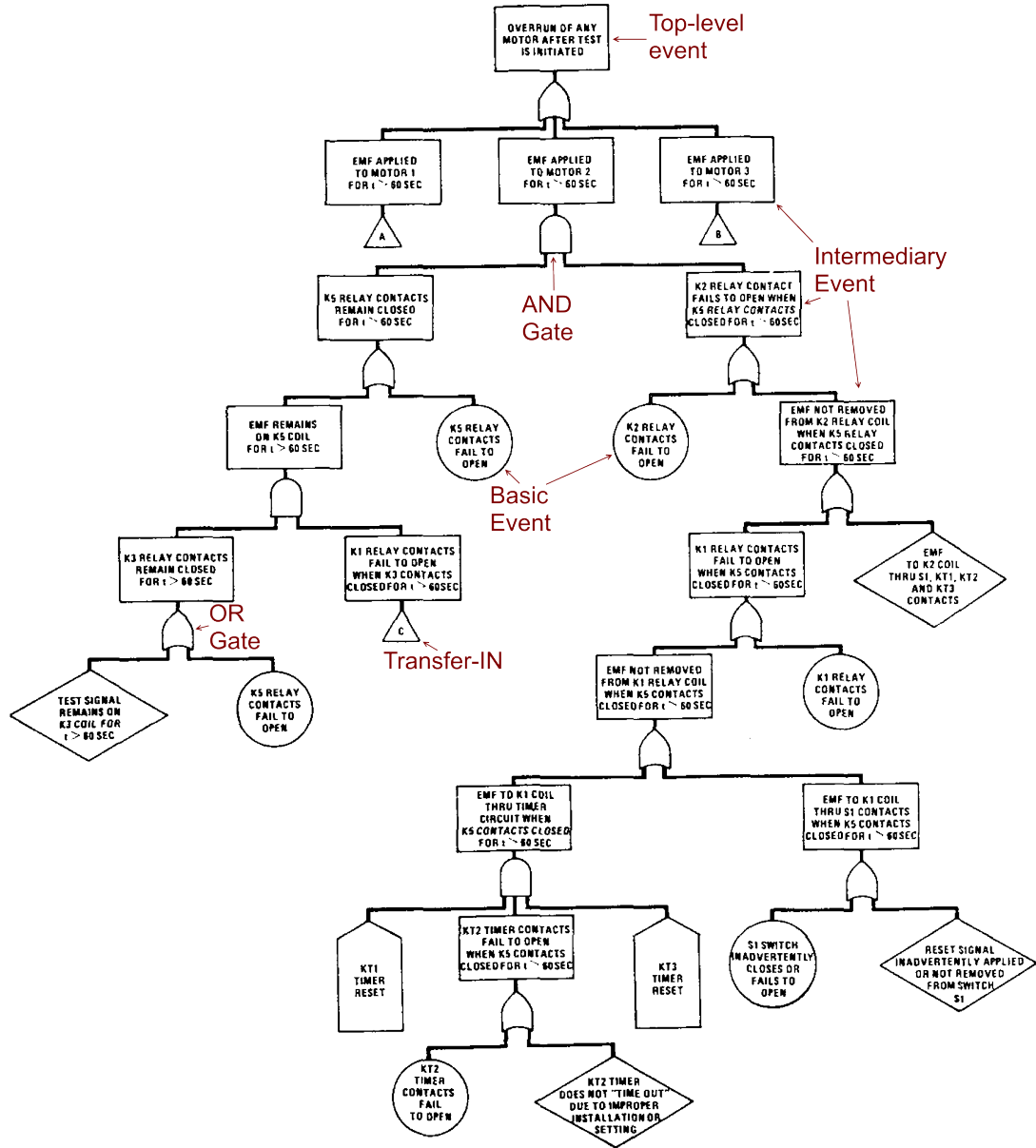


Figure 4 – SFT symbols as in the Fault Tree Handbook

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 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 6 is  $TOP = A \wedge B$ . A technique called *BDD*—which derives from Shannon's method—is explained in Subsection 3.2.2.

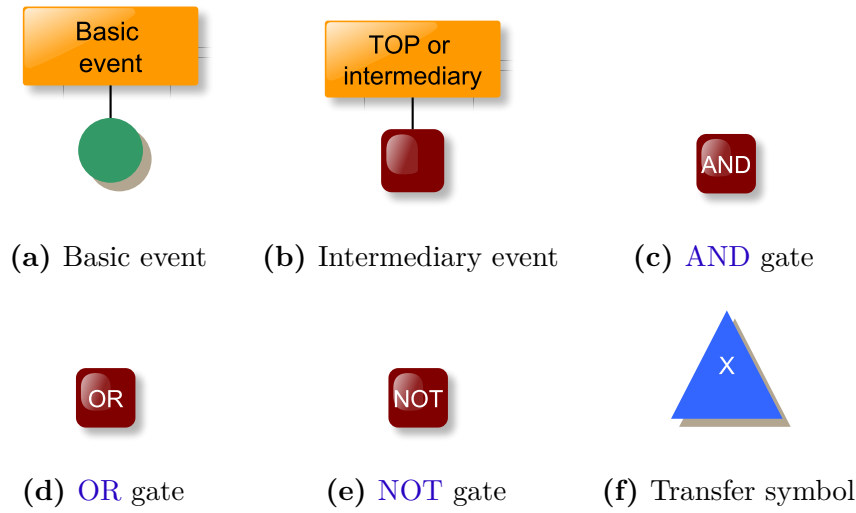


Figure 5 – SFT gates

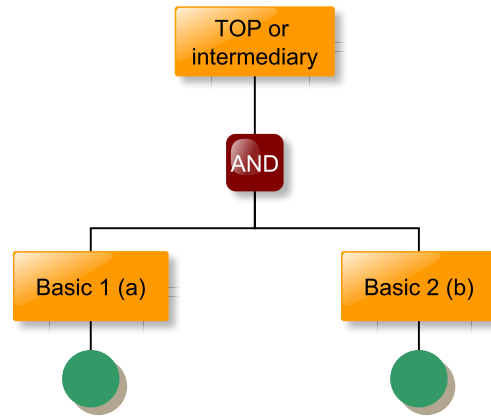


Figure 6 – Very simple example of a fault tree

### 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 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<sup>3</sup> 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

<sup>3</sup> In formulas, the following symbols are used to represent the operators, respectively: “<”, “|”, and “&”

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 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:

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

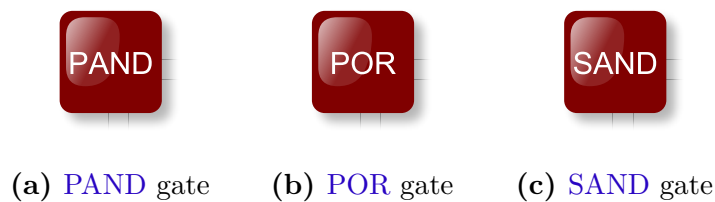
$$S(A \vee B) = \begin{cases} \min(S(A), S(B)) & \text{if } S(A) > 0 \wedge S(B) > 0 \\ \max(S(A), S(B)), & \text{otherwise} \end{cases} \quad (3.1b)$$

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

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

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

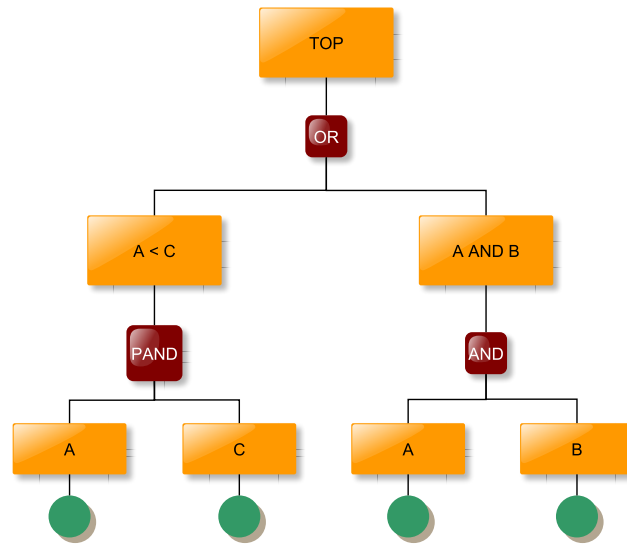
Figure 7 shows **TFT**-specific symbols used in this work. To illustrate **TFTs**, for the formula  $(A < C) \vee (A \wedge B)$ , we show: (i) the **TFT** in Figure 8, and (ii) its corresponding **TTT** in Table 2 (the column '#' indicates the **MCSeq** number).



**Figure 7** – **TFT**-specific gates

**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

**Figure 8** – TFT small example

From structure expressions in order-sensitive FTs (TFT and DFT), MCSeqs are obtained. Several approaches represent MCSeq ordering differently. For the best of our knowledge they are introduced in the work [64] similarly to MCS, allowing set elements with arrows (“ $\rightarrow$ ”) 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  $(X \& Y) | Z$  is a temporal relation (POR) of a temporal relation (SAND). To extract MCSeqs it needs to be converted to  $[X \& Y] \wedge [X | Z] \wedge [Y | Z]$  (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



**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	<b>0</b>
02	0	0	1	0	0	<b>0</b>
03	0	1	0	0	0	<b>0</b>
04	0	1	1	0	0	<b>0</b>
05	0	1	2	0	0	<b>0</b>
06	0	2	1	0	0	<b>0</b>
07	1	0	0	0	0	<b>0</b>
08	1	0	1	0	0	<b>0</b>
09	1	0	2	2	0	<b>2</b>
10	1	1	0	0	1	<b>1</b>
11	1	1	1	0	1	<b>1</b>
12	1	1	2	2	1	<b>1</b>
13	1	2	1	0	2	<b>2</b>
14	1	2	2	2	2	<b>2</b>
15	1	2	3	3	2	<b>2</b>
16	1	3	2	2	3	<b>2</b>
17	2	0	1	0	0	<b>0</b>
18	2	1	0	0	2	<b>2</b>
19	2	1	1	0	2	<b>2</b>
20	2	1	2	0	2	<b>2</b>
21	2	1	3	3	2	<b>2</b>
22	2	2	1	0	2	<b>2</b>
23	2	3	1	0	3	<b>3</b>
24	3	1	2	0	3	<b>3</b>
25	3	2	1	0	3	<b>3</b>

expressions is achieved using DT<sup>4</sup>. 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, 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:

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

$$(X | Y) \vee (X \& Y) \vee (Y | X) = X \vee Y \quad \text{Disjunctive Completion Law} \quad (3.2b)$$

$$(X | Y) \vee (X \& Y) \vee (Y < X) = X \quad \text{Reductive Completion Law 1st} \quad (3.2c)$$

$$(X \wedge Y) \vee (X | Y) = X \quad \text{Reductive Completion Law 2nd} \quad (3.2d)$$

### 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 9 depicts the original gate symbols as shown in [1, 2]. In this work, we use gate symbols as depicted in Figure 10. The informal semantics of them are:

<sup>4</sup> AD Note: Cite dependency tree

**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**.

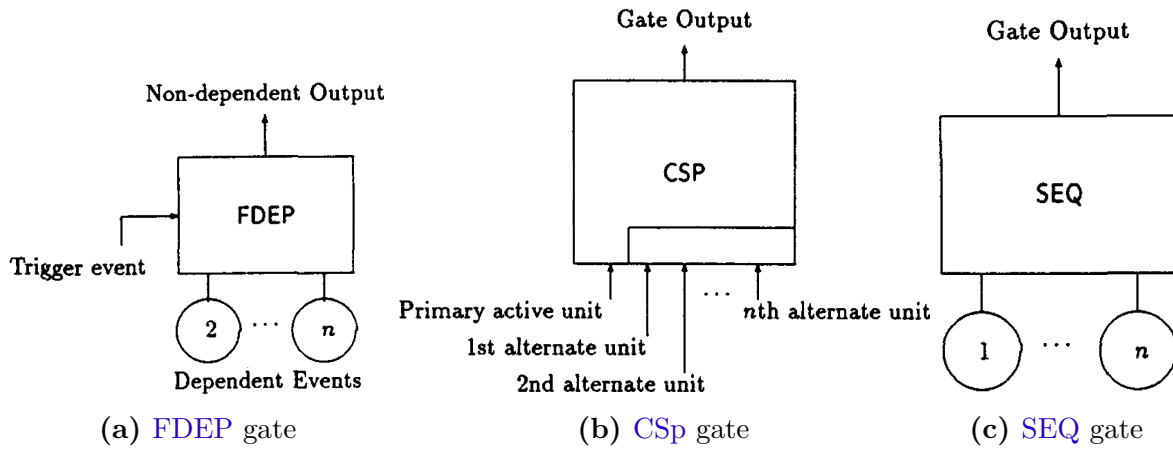


Figure 9 – DFTs's original gates symbols

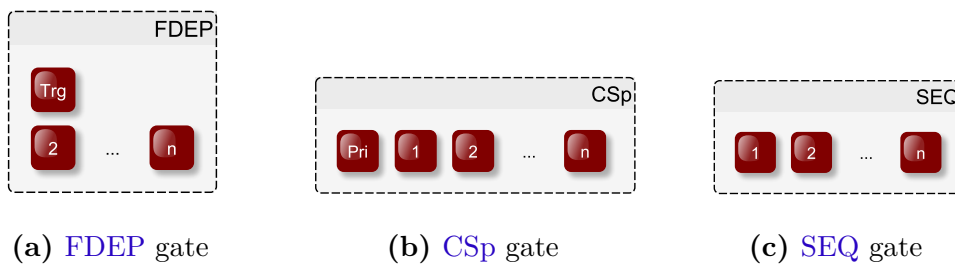


Figure 10 – DFTs's gates symbols

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

**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]

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 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

**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	$(B_a \wedge (A \triangleleft B_a)) \vee (A \wedge (B_d \triangleleft A))$	$A$ is the active input, and $B$ is the spare. Subscripts $a$ and $d$ represent component's state— <i>active</i> and <i>dormant</i> , respectively, which are used on the failure distribution formulas
PAND	$B \wedge (A \trianglelefteq B)$	No distinction of active or dormant states.

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

Operator	Expression	Expr. value if $d(a) < d(b)$	Expr. value if $d(a) = d(b)$	Expr. value if $d(a) > d(b)$
OR	$d(a \vee b)$	$d(a)$	$d(a)$	$d(b)$
AND	$d(a \wedge b)$	$d(b)$	$d(a)$	$d(a)$
NIBefore	$d(a \triangleleft b)$	$d(a)$	$+\infty$	$+\infty$
SIMLT	$d(a \triangle b)$	$+\infty$	$d(a)$	$+\infty$
IBefore	$d(a \trianglelefteq b)$	$d(a)$	$d(a)$	$+\infty$

uses algebraic laws as, for example:

$$(a \triangleleft b) \vee (a \triangle b) \vee (b \triangleleft a) = a \vee b \quad (3.3a)$$

$$(a \wedge (b \triangleleft a)) \vee (a \triangle b) \vee (b \wedge (a \triangleleft b)) = a \wedge b \quad (3.3b)$$

$$(a \trianglelefteq b) \wedge (b \trianglelefteq a) = a \triangle b \quad (3.3c)$$

Figure 11 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, triggers a CPU unit failure. The primary CPU (P) has a warm<sup>5</sup> 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<sup>6</sup>). The top-level event structure expression is:

$$SYSTEM = CS \vee SS \vee (M \wedge MC) \vee \quad (3.4)$$

$$(P \wedge (B_d \triangleleft P)) \vee (B_a \wedge (P \triangleleft B_a)) \vee$$

$$(BP_a \wedge (P2 \triangleleft P1) \wedge (P1 \triangleleft BP_a)) \vee (P2 \wedge (P1 \triangleleft BP_a) \wedge (BP_a \triangleleft P2))$$

<sup>5</sup> Warm spare gates only differ from CSp on the activation time.

<sup>6</sup> Although the original example uses a PAND gate, accordingly to the informal description, a SEQ gate would fit better.



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)<sup>7</sup> 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-suppressed 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).

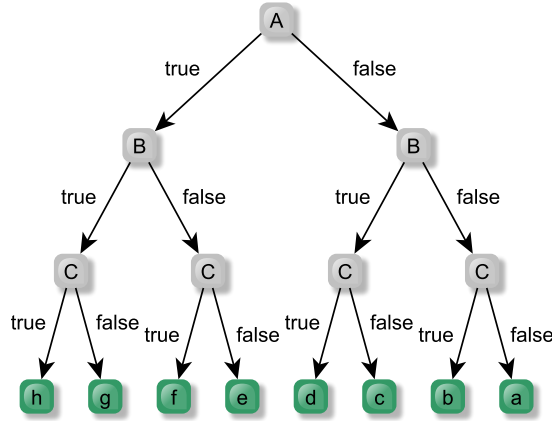
---

<sup>7</sup> AD Note: Cite FSM

**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

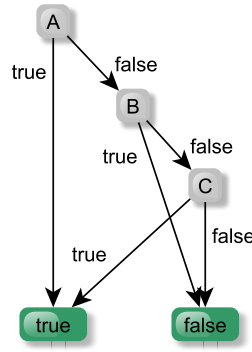
To demonstrate the expressiveness of a **BDD**, Figure 12 shows a diagram for a truth table with three variables (Table 6). In a node, when a path is chosen, the variable of the node assumes the edge value. For example, the top-level node variable of Figure 12 is *A*. Following the right-hand side of 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.

**Figure 12** – A diagram for a truth table

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 *C*, the resulting **BDD** with the binary values nodes (called sink nodes “false” and “true”) for this formula is depicted in Figure 13. 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.

Figure 13 is an **ROBDD**. To be considered an **ROBDD**, the **BDD** must meet the following constraints [78]:

- a) the variables are assigned a constant ordering;



**Figure 13** – A BDD for the expression  $A \vee (\neg B \wedge C)$

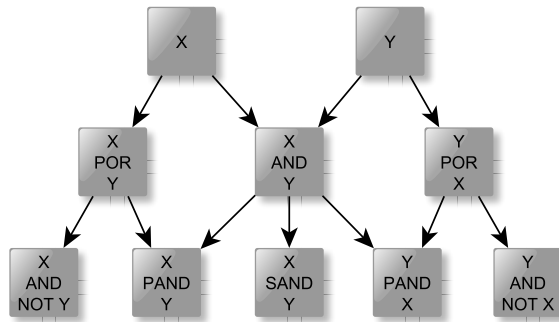
- b) every path to sink nodes visit the input variables in ascending order;
- c) each node represents a distinct logic function.

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 13 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 14 shows a DT with all nodes for variables  $X$  and  $Y$ .

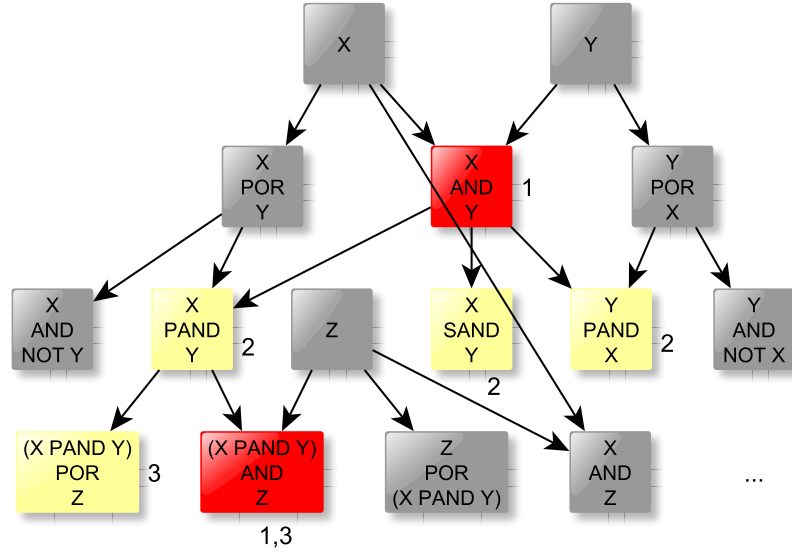


**Figure 14** – DT for variables  $X$  and  $Y$

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 section, we create the **DT** depicted in Figure 15. 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  $X \wedge Y$ .



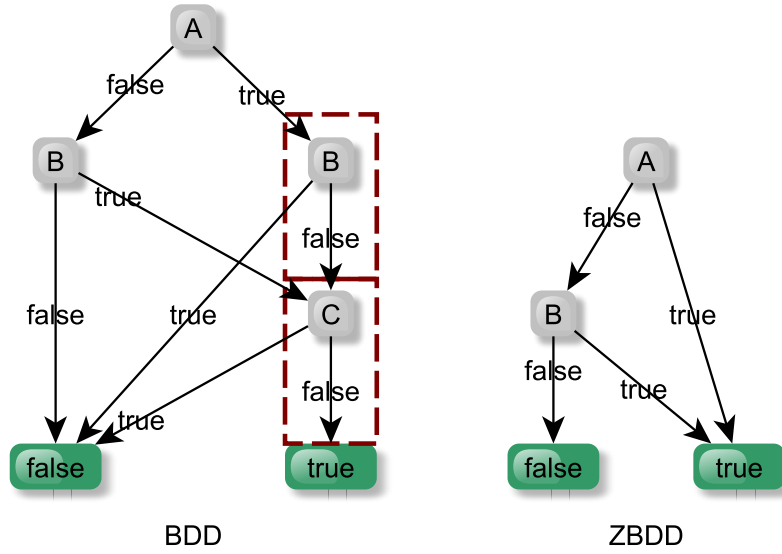
**Figure 15** – **DT** for the formula  $(X \wedge Y) \vee ((X < Y) \wedge Z)$

### 3.2.4 Zero-suppressed Binary Decision Diagrams

The work reported in [64] proposes Zero-suppressed 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 16 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**,



**Figure 16** – ZBDD example of combination set  $\{a, b\}$

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.

### 3.2.5 Sequential Binary Decision Diagrams

SBDD is an extension of BDD to tackle ordering of events in DFTs for CS<sub>p</sub> and WSp gates. Ordering of events in CS<sub>p</sub> gates [43] is slightly different from WSp [42]. A backup system in CS<sub>p</sub> gets activated slower than in WSp, which implies that there are less failure possibilities in CS<sub>p</sub>, but its 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 CS<sub>p</sub>, 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) CS<sub>p</sub> or WSp DFT conversion, and (ii) SBDD model generation. In (i), it is a DFT-to-DFT conversion. CS<sub>p</sub> and WSp gates are converted to a new, but equivalent DFT without CS<sub>p</sub> 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,  $\perp$ , and  $\top$ ) are derived from the previous two operators:

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

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

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

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:  $\{\}, \{\{\}\}, \{\{\}, \{a\}\}, \{\{\}, \{b\}\}, \dots, \{\{a\}, \{a, b\}\}, \dots, \{\{b\}, \{a, b\}\}, \dots, \{\{\}, \{a\}, \{b\}, \{a, b\}\}$ .

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

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

$$\mathbf{var} s = \{X | s \in X\} \implies \mathbf{var} s \in B \quad (\text{variable}) \quad (3.5a)$$

$$X \in B \implies -X \in B \quad (\text{complement}) \quad (3.5b)$$

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

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  $p : B \times \{0, 1\} \rightarrow B$ , such that:

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

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

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

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:*

$$\mathbf{var} i = \mathbf{var} j \iff i = j \quad (3.8a)$$

$$\mathbf{var} i \neq -\mathbf{var} j \quad (3.8b)$$

$$-\mathbf{var} i \neq \mathbf{var} j \quad (3.8c)$$

### 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  $\Phi(x) : B^n \rightarrow B^1$  be a binary function of a vector of binary variables, such that  $x = [x_1, x_2, \dots, x_n]$ , 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  $y = [y_1, y_2, \dots, y_n]$ , monotonicity of  $\Phi$  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:  $x[x_i/a] = [x_1, \dots, x_{i-1}, a, x_{i+1}, \dots, x_n]$

### 3.4.1 Non-coherent fault tree misleads

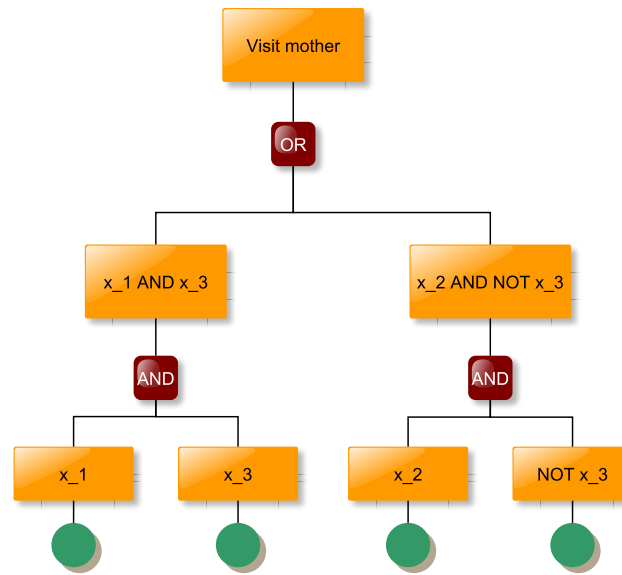
In this section we illustrate—with the second example detailed in [11]—how non-coherent FT misleads.

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 17. It is clear that the structure function is non-coherent in  $x_3$  accordingly to Definition 3.3:  $\Phi(1, 1, x_3)[x_3/1] = \Phi(1, 1, x_3)[x_3/0]$ .

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

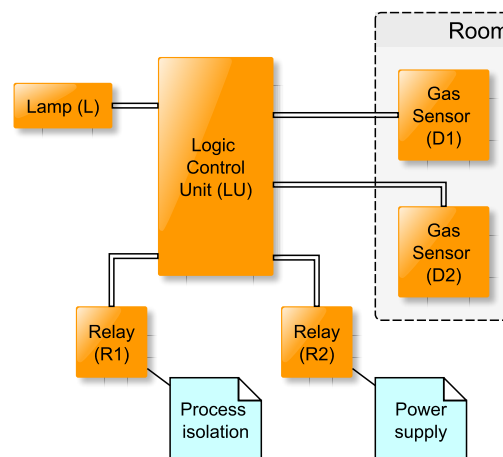
In this section we show the second example detailed in [9].



**Figure 17** – Non-coherent FT college student’s example

The gas detection system depicted in Figure 18 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:

- shuts-down the main system (process isolation) by de-energizing relay  $R_1$ ;
- informs the operator of the leakage by lamp and siren  $L$ ;
- deactivates all possible ignition sources, which is the interruption of power supply by de-energizing relay  $R_2$ .



**Figure 18** – Gas detection system

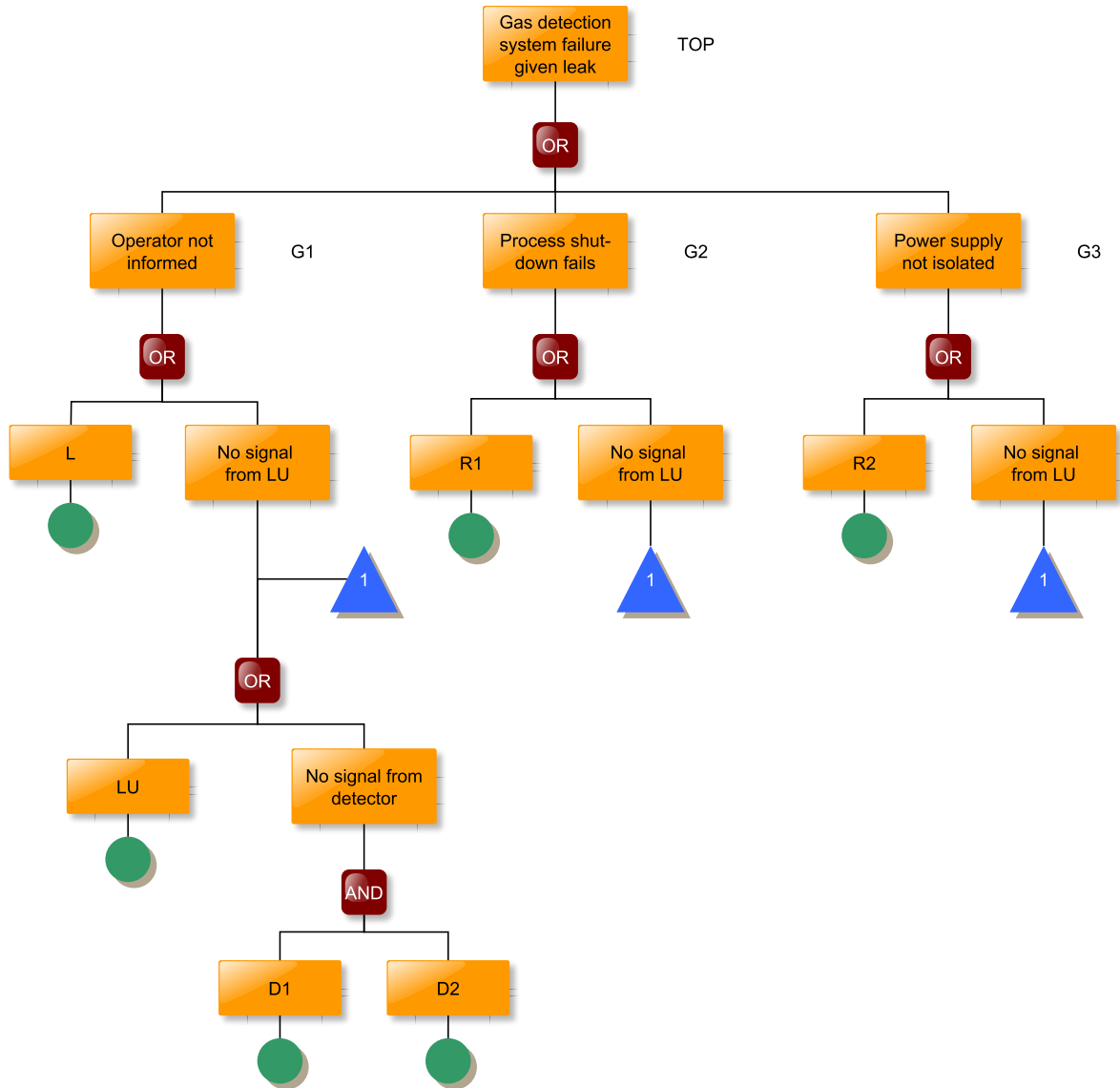
The system is in a fail state if it does not perform one of these three tasks. The fault tree that represents this generic failure is depicted in Figure 19.  $G_1$ ,  $G_2$ , and  $G_3$  are subtrees that represents the three tasks “Operator not informed”, “Process shut-down fails”, and

“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:

$$G_1 = L \vee LU \vee (D_1 \wedge D_2)$$

$$G_2 = R_1 \vee LU \vee (D_1 \wedge D_2)$$

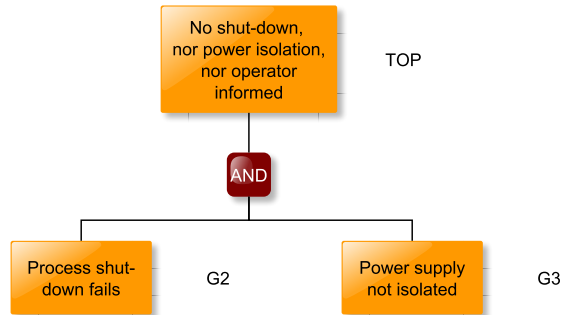
$$G_3 = R_2 \vee LU \vee (D_1 \wedge D_2)$$



**Figure 19** – FT for a generic failure in the gas detection system

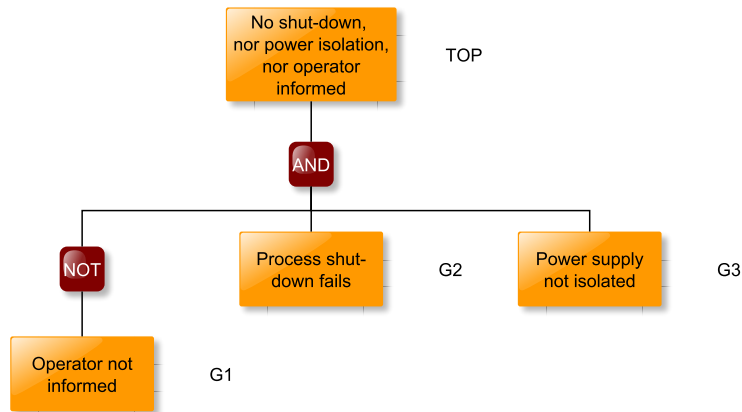
Analysing in more detail, there are different degrees of system failure. There are eight outcomes (given the three tasks) and the most critical one is when both process shut-down ( $G_2$ ) and power supply isolation ( $G_3$ ) fail keeping energized upon a leakage, and the operator is not informed ( $G_1$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

in Figure 20. The minimal cut sets obtained from this will be:  $\{R_1, R_2\}$ ,  $\{D_1, D_2\}$ , and  $\{LU\}$ .



**Figure 20** – *Coherent FT* for the most critical outcome of the gas detection system

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 21.



**Figure 21** – *Non-coherent FT* for the most critical outcome of the gas detection system

If the operator *can* be informed, then cut sets  $\{D_1, D_2\}$  and  $\{LU\}$  could not have occurred (see Figure 19), thus the correct qualitative analysis should consider only cut set  $\{R_1, R_2\}$ . Reducing the expressions of the non-coherent FT (Figure 21), 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 quantitative analysis.<sup>8 9</sup>

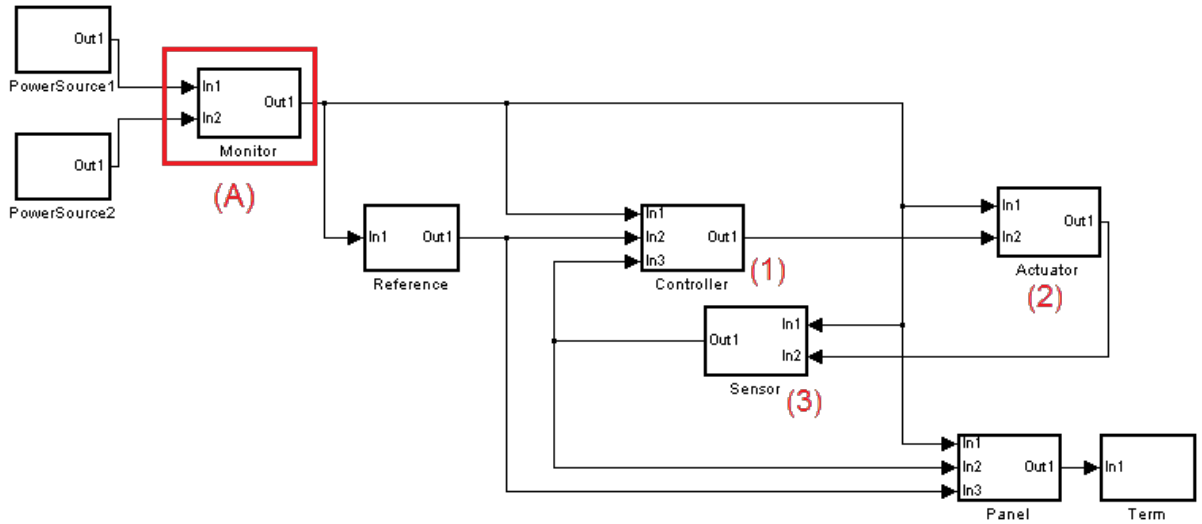
<sup>8</sup> AM Note: Isto poderia ser argumentado pro exemplo anterior? Seria legal ter o mesmo exemplo e pontos de vista distintos (não coerente e coerente).

<sup>9</sup> AD Note: Eu apenas reporte o exemplo. Essa análise foi o autor do exemplo quem fez. Se der vou acrescentar para a outra, não deve ser muito complicado... preciso de mais tempo apenas para analisar com calma.



### 3.5 Systems nominal model and fault injection

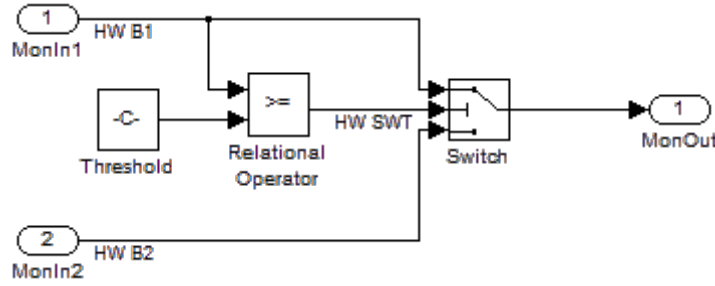
Control system modelling using Simulink block diagrams [82] is recommended in [30] and have been used by our industrial partner. It is a complementary tool of Matlab [83]. In fact, it works as a graphical interface to Matlab. A Simulink model has blocks and connections between these blocks, named signals. 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.



**Figure 22** – Block diagram of the ACS provided by EMBRAER (nominal model)

Nowadays, control systems are usually composed of an electromechanical part and a processor. Figure 22 shows the components of a feedback system [84] which was provided by EMBRAER. 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.

Figure 23 shows the internal elements of the monitor component (Figure 22 (A)), which is used as a case study in Chapter 6 to illustrate our strategy. The outputs of the



**Figure 23** – Internal diagram of the monitor component (Figure 22 (A)).

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<sub>M</sub>*. 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<sub>M</sub>* 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<sub>M</sub>* language. The resulting *CSP<sub>M</sub>* code (implementation) is then used to check if it meets functional requirements also encoded in *CSP<sub>M</sub>* (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<sub>M</sub>* 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<sub>M</sub>* 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 23, some representative fault traces are:

```
TRACE 1:
failure.Hardware.NO4_RelationalOperator.1.EXP.B.true
failure.Hardware.NO4_RelationalOperator.1.ACT.B.false
failure.Hardware.NO4_MonIn2.1.EXP.I.5
failure.Hardware.NO4_MonIn2.1.ACT.OMISSION
out.1.OMISSION
```

```
TRACE 2:
failure.Hardware.NO4_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

```

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 eight 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<sub>M</sub>** 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 EMBRAER

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:

$$\text{TRACE 1} = S \wedge B$$

$$\text{TRACE 2} = B \wedge S$$

$$\text{TRACE 3} = A \wedge B$$

$$\text{TRACE 4} = B \wedge A$$

$$\text{TRACE 5} = A \wedge S$$

$$\text{TRACE 6} = A \wedge S \wedge B$$

$$\text{TRACE 7} = A \wedge B \wedge S$$

$$\text{TRACE 8} = B \wedge A \wedge S$$

And we combine them as a single Boolean expression:  $\text{TRACE 1} \vee \text{TRACE 2} \vee \text{TRACE 3} \vee \text{TRACE 4} \vee \text{TRACE 5} \vee \text{TRACE 6} \vee \text{TRACE 7} \vee \text{TRACE 8}$ , 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 EMBRAER if we use the following association (Table 7):

$$A = \text{LowPower-In1}$$

$$B = \text{LowPower-In2}$$

$$S = \text{SwitchFailure}$$

Note that when we combine each fault with **AND** gates, we lose the information about order<sup>10</sup>:  $S \wedge B$  and  $B \wedge S$  are equal, due to the commutative law of Boolean expressions.

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

<sup>10</sup> In our previous work we designed the observer to ignore order as well, by making similar traces—with different ordering—the same size. Here we modified the observer specification to make similar traces with different sizes.

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<sup>11</sup>:

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<sup>12</sup> 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<sup>13</sup>, which provides a text editor, syntax parser, shortcuts, etc. (see Figure 24).

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).

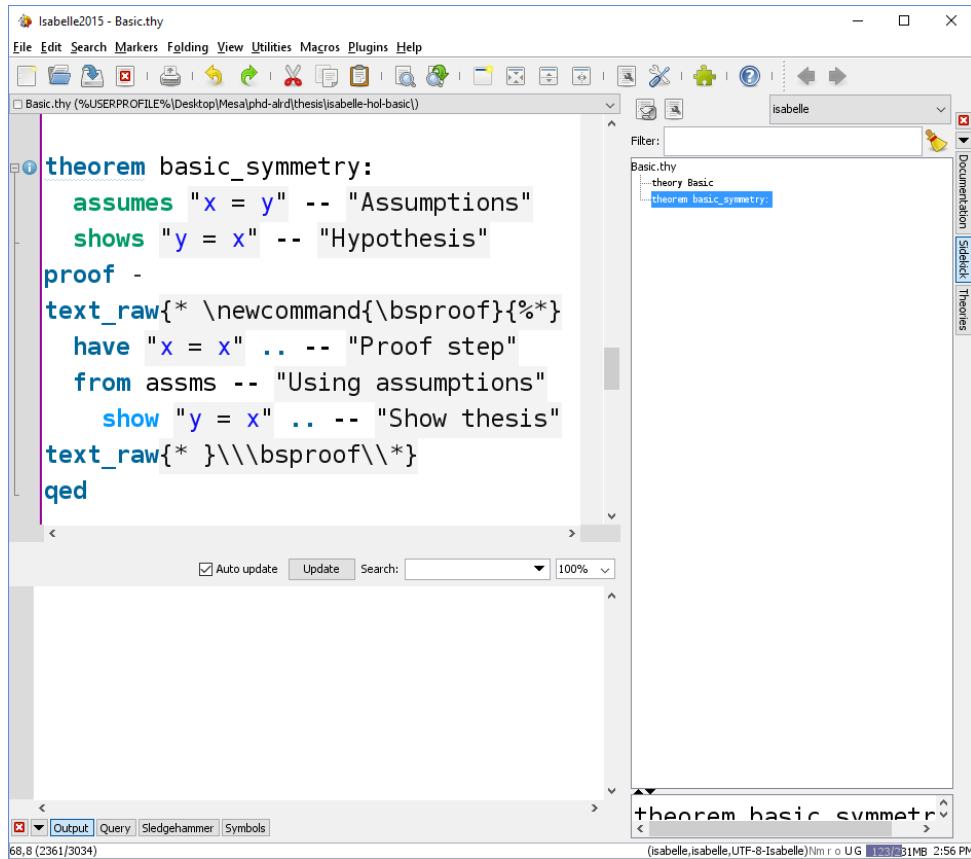
Besides the provided theories, its active community provides a comprehensive archive of formal proofs<sup>14</sup> (AFP). Each entry in this archive can be cited and usually contains an *abstract*, a document, and a theory file. For example, a Free Boolean Algebra

<sup>11</sup> Accessed 27/jan/2016: <<https://isabelle.in.tum.de/overview.html>>

<sup>12</sup> By human we mean that anyone with mathematics and logic basic knowledge—it means that deep programming knowledge is not essential.

<sup>13</sup> Accessed 27/jan/2016: <<http://www.jedit.org/>>

<sup>14</sup> Accessed 27/jan/2016: <<http://afp.sourceforge.net/>>



**Figure 24** – Isabelle/HOL window, showing the basic symmetry theorem

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

**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](#).

[15](#) [16](#)

---

<sup>15</sup> AD Note: Use BA1993 to minimise the gap between systems models and fault trees. Also: MCS+1999

<sup>16</sup> AD Note: Write the problem: simple mathematical notation for all fault trees. Include direct DFT-to-ATF mapping.



## 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. 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<sup>1</sup>). 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.

<sup>1</sup> 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.

De August  
tulo é desr  
mente con  
palmente p  
da contrib  
importante  
final, ilustr  
ceito e pro  
exemplos.  
destacar re  
portantes,  
da mecani  
abelle. Dev  
seção ou u  
dedicado à  
ção, não in  
obviamente  
trando um  
exemplo.”

Explain th  
of (i) fault  
(ii) theore  
and (iii) sy  
cution.

dizer que n  
dagem usa  
dagem sim  
expressões  
mas que te  
de dar um  
denotacion  
em conjun

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.

The **ATF** form a free algebra, similarly to **FBA**s. *Infimum* and *Supremum* are defined as set intersection ( $\cap$ ) and union ( $\cup$ ) respectively. The order within the algebra is defined 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]:

$$S \rightarrow T = \{zs | \exists xs, ys \bullet (\mathbf{set} \ xs) \cap (\mathbf{set} \ ys) = \{\} \wedge xs \in S \wedge ys \in T \wedge zs = xs @ ys\} \quad (4.1)$$

where the **set** function returns the set of the elements of a list, and  $@$  concatenates two lists.

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:

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

$$xs_{[..j]} = xs_{[0..j]} \quad (4.2b)$$

$$xs_{[i..]} = xs_{[i..|xs|]} \quad (4.2c)$$

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

$$xs_{[i..j]} = xs_{[L..U]} \quad (4.3)$$

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:

$$\forall i \bullet xs_{[..i]} @ xs_{[i..]} = xs \quad (4.4)$$

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

$$S \rightarrow T = \{zs | \exists i \bullet zs_{[..i]} \in S \wedge zs_{[i..]} \in T\} \quad (4.5)$$

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

$$\mathbf{var} x = \{zs | x \in zs\} \quad (4.6)$$

where *set zs* is the set of the elements of *zs*, and  $x \in zs$  is defined as  $x \in \text{set } zs$

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

$$\mathbf{var} x \in \mathbf{atf} \quad \text{Variable} \quad (4.7a)$$

$$S \in \mathbf{atf} \implies -S \in \mathbf{atf} \quad \text{Complement, Negation} \quad (4.7b)$$

$$S \in \mathbf{atf} \wedge T \in \mathbf{atf} \implies S \cap T \in \mathbf{atf} \quad \text{Intersection, Infimum} \quad (4.7c)$$

$$S \in \mathbf{atf} \wedge T \in \mathbf{atf} \implies S \rightarrow T \in \mathbf{atf} \quad \text{XBefore} \quad (4.7d)$$

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

$$\text{UNIV} \in \mathbf{atf} \quad \text{Universal set, True} \quad (4.7e)$$

$$\{\} \in \mathbf{atf} \quad \text{Empty set, False} \quad (4.7f)$$

$$S \in \mathbf{atf} \wedge T \in \mathbf{atf} \implies S \cup T \in \mathbf{atf} \quad \text{Union, Supremum} \quad (4.7g)$$

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

$$\mathbf{var} a = \mathbf{var} b \iff a = b \quad (4.8a)$$

$$\mathbf{var} a \not\subseteq -\mathbf{var} b \quad (4.8b)$$

$$\mathbf{var} a \neq -\mathbf{var} b \quad (4.8c)$$

$$-\mathbf{var} a \not\subseteq \mathbf{var} b \quad (4.8d)$$

$$-\mathbf{var} a \neq \mathbf{var} b \quad (4.8e)$$

Expressions (4.7a) to (4.7g) and (??) to (4.8e) implies that the **ATF** without the **XBefore** operator (4.1) forms a Boolean algebra based on sets of lists. And this is also equivalent to an **FBA** with the same generators.

In our previous work [86] we stated a relation of **XBefore** and *supremum*, provided the operands are variables (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 (4.6).

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).

$$\mathbf{tempo}_1 S = \forall i, j, zs \bullet i \leq j \implies \neg (zs_{[..i]} \in S \wedge zs_{[j..]} \in S) \quad (4.9a)$$

$$\mathbf{tempo}_2 S = \forall i, zs \bullet zs \in S \iff zs_{[..i]} \in S \vee zs_{[i..]} \in S \quad (4.9b)$$

$$\mathbf{tempo}_3 S = \forall i, j, zs \bullet j < i \implies (zs_{[j..i]} \in S \iff zs_{[..i]} \in S \wedge zs_{[j..]} \in S) \quad (4.9c)$$

$$\mathbf{tempo}_4 S = \forall zs \bullet zs \in S \iff (\exists i \bullet zs_{[i..(i+1)]} \in S) \quad (4.9d)$$

The second temporal property is about belonging to one sublist in the beginning or in the end. If a generator is in a list, then it must be at the beginning or at the end.

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).

Finally, if a generator belongs to a list, then there is a sublist of size one that contains the generator.

Variables have all four temporal properties. For a generator  $x$ , the following is valid:

$$\mathbf{tempo}_1(\mathbf{var} x) \wedge \mathbf{tempo}_2(\mathbf{var} x) \wedge \mathbf{tempo}_3(\mathbf{var} x) \wedge \mathbf{tempo}_4(\mathbf{var} x) \quad (4.10)$$

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 (4.1):

$$S \rightarrow T = \{zs \mid \exists xs, ys \bullet xs \in S - T \wedge ys \in T - S \wedge \text{distinct } zs \wedge zs = xs @ ys\} \quad (4.11)$$

Other expressions also meet one or more temporal properties:

$$\mathbf{tempo}_1 S \wedge \mathbf{tempo}_1 T \implies \mathbf{tempo}_1 (S \cap T) \quad (4.12a)$$

$$\mathbf{tempo}_3 S \wedge \mathbf{tempo}_3 T \implies \mathbf{tempo}_3 (S \cap T) \quad (4.12b)$$

$$\mathbf{tempo}_2 S \wedge \mathbf{tempo}_2 T \implies \mathbf{tempo}_2 (S \cup T) \quad (4.12c)$$

$$\mathbf{tempo}_4 S \wedge \mathbf{tempo}_4 T \implies \mathbf{tempo}_4 (S \cup T) \quad (4.12d)$$

## 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 **XBefore**s 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**:

$$\begin{aligned} & (A \cap -B) \cup ((A \rightarrow B) \cap C) \\ & A \cup B \\ & A \rightarrow B \\ & A \cap B \\ & A \rightarrow B \rightarrow C \end{aligned}$$

The following formulas are *not* in **DNF**:

$$\begin{aligned} & -(A \cup B) \\ & A \cap (B \cup C) \\ & A \rightarrow (B \cup C) \\ & A \rightarrow (B \cap C) \end{aligned}$$

But to transform the last two formulas into **DNF**, one can use Laws (4.16a), (4.16b), (4.16c) and (4.16d), for instance.

We define events independence ( $\triangleleft$ ) as the property that one operand does not imply the other. For example, we need to avoid that the operands of **XBefore** are **var**  $a$  and **var**  $a \cup \text{var } b$  (it results in  $\{\}$ , see (4.14e)).

$$S \triangleleft T = \forall i, z_s \bullet \neg (zs_{[i..(i+1)]} \in S \wedge zs_{[i..(i+1)]} \in T) \quad (4.13)$$

The absence of occurrences ( $\{\}$ , the empty set of **atf**) is a “0” for the **XBefore** operator.

$$\{\} \rightarrow S = \{\} \quad \text{left-false-absorb} \quad (4.14a)$$

$$S \rightarrow \{\} = \{\} \quad \text{right-false-absorb} \quad (4.14b)$$

$$(S \rightarrow T) \cup S = S \quad \text{left-union-absorb} \quad (4.14c)$$

$$(T \rightarrow S) \cup S = S \quad \text{right-union-absorb} \quad (4.14d)$$

$$\mathbf{tempo}_1 S \implies S \rightarrow S = \{\} \quad \text{non-idempotent} \quad (4.14e)$$

$$\mathbf{tempo}_1 S \wedge \mathbf{tempo}_1 T \wedge \mathbf{tempo}_1 U \implies$$

$$S \rightarrow (T \rightarrow U) = (S \rightarrow T) \rightarrow U \quad \text{associativity} \quad (4.14f)$$

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 union and intersection.

$$\begin{aligned} \mathbf{tempo}_1 S \wedge \mathbf{tempo}_1 T &\implies \\ (S \rightarrow T) \cap (T \rightarrow S) &= \{\} \quad \text{inter-equiv-false} \end{aligned} \quad (4.15a)$$

$$\begin{aligned} \mathbf{tempo}_{1-4} S \wedge \mathbf{tempo}_{1-4} T \wedge S \triangleleft T &\implies \\ (S \rightarrow T) \cup (T \rightarrow S) &= S \cap T \quad \text{union-equiv-inter} \end{aligned} \quad (4.15b)$$

As **XBefore** is not symmetric, the intersection of symmetrical **XBefores** is empty. The union of symmetrical **XBefores** is a partition of the intersection of the operands.

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

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

$$(S \cup T) \rightarrow U = (S \rightarrow U) \cup (T \rightarrow U) \quad \text{left-union-dist} \quad (4.16a)$$

$$S \rightarrow (T \cup U) = (S \rightarrow T) \cup (S \rightarrow U) \quad \text{right-union-dist} \quad (4.16b)$$

$$\begin{aligned} \mathbf{tempo}_{1-4} S \wedge \mathbf{tempo}_{1-4} T \wedge S \triangleleft T &\implies \\ (S \cap T) \rightarrow U &= (S \rightarrow T \rightarrow U) \cup \\ &\quad (T \rightarrow S \rightarrow U) \quad \text{left-inter-dist} \end{aligned} \quad (4.16c)$$

$$\begin{aligned} \mathbf{tempo}_{1-4} T \wedge \mathbf{tempo}_{1-4} U \wedge T \triangleleft U &\implies \\ S \rightarrow (T \cap U) &= (S \rightarrow T \rightarrow U) \cup \\ &\quad (S \rightarrow U \rightarrow T) \quad \text{right-inter-dist} \end{aligned} \quad (4.16d)$$

$$\begin{aligned} \mathbf{tempo}_2 S &\implies S \cap (T \rightarrow U) = ((S \cap T) \rightarrow U) \cup \\ &\quad (T \rightarrow (S \cap U)) \quad \text{unordered} \end{aligned} \quad (4.16e)$$

**XBefore** is distributive over union. On the other hand, the intersection is related to order. Thus it is not distributive with **XBefore**. Finally, the intersection of an event with an **XBefore** states that such an event can occur in any order within the events in the **XBefore**.

The law name, unordered, of (4.16e) is clearer if we expand (4.16e) with (4.16c) and (4.16d):

$$\begin{aligned} \mathbf{tempo}_{1-4} S \wedge \mathbf{tempo}_{1-4} T \wedge \\ \mathbf{tempo}_{1-4} U \wedge S \triangleleft T \wedge S \triangleleft U &\implies \\ S \cap (T \rightarrow U) &= (S \rightarrow T \rightarrow U) \cup \\ &\quad (T \rightarrow S \rightarrow U) \cup \\ &\quad (T \rightarrow U \rightarrow S) \quad \text{expanded-unordered} \end{aligned} \quad (4.17)$$

### 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 quantity of faults in the minimal sequences that cause the root failure, and (ii) root probability, provided the probability of sequences occurrences. These properties are representative and other criteria can be modelled similarly.

The denotational semantics of ATF is a set of distinct lists, thus each list has no repeated elements and each list represent a possible combination of faults that causes the root failure. MCSeqs are the lists with least length, thus have minimal lengths of the lists.

**Definition 4.1** (Minimal cut sequences). Let  $F$  be a formula in ATF and  $L$  the denotational semantics of  $F$ , the minimal cut sequences MCSeq of  $F$  are:

$$\text{MCSeq}(F) = \{xs | xs \in L \bullet |xs| = \min L\} \quad (4.18)$$

where

$$\min L = \text{Min}(\{|xs| | xs \in L\})$$

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.

$$\begin{aligned} P(t) &= \Pr\{T_1 \leq T_2\}^{<t} \\ &= \int_0^t P'_2(x)P_1(x)dx \end{aligned} \quad (4.19)$$

where  $P_i$  is the probability of the occurrence of the  $i$ th fault (cumulative distribution function), and  $T_i$  is the time that the  $i$ th fault occurs. The general case (reported in [87]) of a PAND gate with  $n$  inputs is:


$$\begin{aligned} \Pr\{[f_n, f_{n-1} \dots, f_2, f_1]\} &= \int_0^t P'_1(t_1) \int_0^{t_1} P'_2(t_2) \dots \int_0^{t_{n-2}} P'_{n-1}(t_{n-1}) \\ &\quad \int_0^{t_{n-1}} P'_n(t_n) dt_n dt_{n-1} \dots dt_2 dt_1 \end{aligned} \quad (4.20)$$


where  $f_n$  is the first fault to occur, and  $f_1$  is the last.

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,  $\Pr\{[f_1, f_2]\}$  contains the situations of  $\Pr\{[f_1, f_2, f_3]\}$ . To isolate these cases, we use the filter notation, grouping all lists that contains repeated situations to calculate probability. The probability of a list  $xs$  in the denotational semantics of ATF  $L$  is:

$$\text{DPr}_L\{xs\} = \Pr\{xs\} - \sum_{\{ys | ys \in L - \{xs\} \bullet xs = \text{filter}_{xs}(ys)\}} \Pr\{ys\} \quad (4.21)$$

where  $\text{filter}_{xs}(ys)$  removes all elements from  $ys$  that are not in  $xs$ . For example,  $\text{filter}_{[f_1, f_2]}([f_1, f_3, f_2]) = [f_1, f_2]$ .


We use the traditional **probabilities** calculations [3] as reference, ~~and~~ to calculate probabilities of formulas in **ATF**. 

$$\Pr \{\mathbf{var} f_1 \wedge \mathbf{var} f_2\} = P_1(t) \times P_2(t) \quad (4.22a) \quad \text{$$

$$\Pr \{\mathbf{var} f_1 \vee \mathbf{var} f_2\} = P_1(t) + P_2(t) - P_1(t) \times P_2(t) \quad (4.22b)$$

The probability in a formula in ATF is calculated using its denotational semantics.

For example, the formula  $(\mathbf{var} f_1 \rightarrow \mathbf{var} f_2) \vee (\mathbf{var} f_2 \rightarrow \mathbf{var} f_1)$ , 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):

$$\begin{aligned} \text{DPr}_L \{[f_1, f_2], [f_2, f_1]\} &= \text{DPr}_L \{[f_1, f_2]\} + \text{DPr}_L \{[f_2, f_1]\} - \quad \text{$$
 \\ &\quad \text{DPr}\_L \{[f\_1, f\_2]\} \times \text{DPr}\_L \{[f\_2, f\_1]\} \\ &= \text{DPr}\_L \{[f\_1, f\_2]\} + \text{DPr}\_L \{[f\_2, f\_1]\} \quad \text{ignored } \times \\ &= \Pr \{[f\_1, f\_2]\} + \Pr \{[f\_2, f\_1]\} \quad \text{Eq. (4.21), both } \sum \text{ is } 0 \\ &= \int\_0^t P'\_2(x)P\_1(x)dx + \int\_0^t P'\_1(x)P\_2(x)dx \quad \text{by Eq. (4.19)} \\ &= \int\_0^t (P'\_2(x)P\_1(x) + P'\_1(x)P\_2(x)) dx \quad \text{by sum of } \int \\ &= \int\_0^t (P\_1(x)P\_2(x))' dx \quad \text{by inv. deriv. product} \\ &= P\_1(t) \times P\_2(t) \quad (4.23) \end{aligned}

The term  $\text{DPr}_L \{[f_1, f_2]\} \times \text{DPr}_L \{[f_2, f_1]\}$  can be safely removed because both situations cannot occur simultaneously.

In Eq. (4.23) we demonstrated that the probability of a formula  $(\mathbf{var} f_1 \rightarrow \mathbf{var} f_2) \vee (\mathbf{var} f_2 \rightarrow \mathbf{var} f_1)$  is equal to the probability of the traditional **AND** probability  $(\mathbf{var} f_1 \wedge \mathbf{var} f_2)$ , shown in Eq. (4.22a). This is expected as these two formulas are equivalent, as shown in Eq. (4.15b).

**Definition 4.2** (Probability of a formula in **ATF**). *Let  $F$  be a formula in **ATF**, and  $L$  its denotational semantics, then:*

$$\Pr \{F\} = \sum_{xs \in L} \text{DPr}_L \{xs\} \quad (4.24) \quad \text{$$

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  $\neg \mathbf{var} f_1$  (for generators  $f_1$  and  $f_2$ ) is:

$$\begin{aligned}
 \Pr \{ \neg \mathbf{var} f_1 \} &= \text{DPr}_L \{ [], [f_2] \} \\
 &= \text{DPr}_L \{ [] \} + \text{DPr}_L \{ [f_2] \} && \text{by Eq. (4.24)} \\
 &= \Pr \{ [] \} - \Pr \{ [f_2] \} + \Pr \{ [f_2] \} && \text{by Eq. (4.21)} \\
 &= \Pr \{ [] \} \\
 &\approx 1
 \end{aligned} \tag{4.25}$$

The empty list is the special case to represent the approximation of the probability calculation of NOT operators. 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, thus its probability value depends on the value of the probabilities of all constituents be operational, which is *usually* near 1 (see [9]). This is the only assumption in the calculations shown in this section.

Another example including complement is the formula  $\mathbf{var} f_1 \wedge \mathbf{var} f_2 \wedge \neg \mathbf{var} f_3$ , which has probability  $P_1(t) \times P_2(t)$ :

$$\begin{aligned}
 \Pr \{ \mathbf{var} f_1 \wedge \mathbf{var} f_2 \wedge \neg \mathbf{var} f_3 \} &= \text{DPr}_L \{ [f_1, f_2], [f_2, f_1] \} \\
 &= P_1(t) \times P_2(t) && \text{by Eq. (4.23)}
 \end{aligned} \tag{4.26}$$

Finally, using Eq. (4.24), we demonstrate the equivalence to probability of an OR operator, calculated using the denotational semantics:

$$\begin{aligned}
 \Pr \{ \mathbf{var} f_1 \vee \mathbf{var} f_2 \} &= \text{DPr}_L \{ [f_1], [f_2], [f_1, f_2], [f_2, f_1] \} \\
 &= \Pr \{ [f_1] \} - \Pr \{ [f_1, f_2] \} - \Pr \{ [f_2, f_1] \} + \\
 &\quad \Pr \{ [f_2] \} - \Pr \{ [f_1, f_2] \} - \Pr \{ [f_2, f_1] \} + \\
 &\quad \Pr \{ [f_1, f_2] \} + \Pr \{ [f_2, f_1] \} \\
 &= \Pr \{ [f_1] \} + \Pr \{ [f_2] \} - (\Pr \{ [f_1, f_2] \} + \Pr \{ [f_2, f_1] \}) \\
 &= P_1(t) + P_2(t) - P_1(t) \times P_2(t)
 \end{aligned} \tag{4.27}$$

### 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:

$$|F|^{\leq n} = \text{Min}(\{|xs| \mid xs \in L\}) \leq n \quad \text{length of MCSeqs} \quad (4.28a)$$

$$\Pr\{F\}^{< P_x} = \Pr\{F\} < P_x \quad \text{root-event probability} \quad (4.28b)$$

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

## 4.4 Propositions

In this section we discuss the theorems and definitions that still need to be proved. We present them as propositions.

Soundness and completeness of the ATF is given in terms of the algebraic form and its denotational semantics (Subsection 4.4.1). The AL is defined in terms of a logic that is decidable and some output value (Subsection 4.4.2).

### 4.4.1 Soundness and completeness of ATF

Given the semantics of a formula of ATF, there is always a set of sequences that represents exactly the formula. To guarantee the completeness we show that for every set of sequences there is a corresponding formula in ATF.

**Proposition 4.1** (Soundness and completeness of ATF). <sup>2 3 4</sup> *Let  $F$  be the set of all formulas in ATF, and  $SS$  be the set of all sets of sequences:*

$$\forall f \in F. \exists S \in SS. f = S \quad \text{Soundness} \quad (4.29a)$$

$$\forall S \in SS. \exists f \in F. f = S \quad \text{Completeness} \quad (4.29b)$$

The equality in the proposition is set-based, thus, in both cases,  $f \subseteq S \wedge S \subseteq f$ .

### 4.4.2 AL concepts

The Activation Logic (AL) is used to model system faults. When reasoning about faults, engineers analyse component by component, defining its outputs in the presence of each possible fault. The AL is nothing more than this: the outputs of the components in the presence of (some combination of) faults. To ensure that all possibilities are considered (those that the engineer reasoned about them), the formula that results from the output

<sup>2</sup> AM Note: Está de acordo com os livros de lógica?

<sup>3</sup> AD Note: Não tive muito tempo para pensar, mas sim. Eu olhei o significado de ambos e montei a proposição.

<sup>4</sup> AS Note: A definição de completess é intuitiva, mas a de soundness, não. Esperaria que fosse algo como a equivalência de duas formulas estivesse relacionada à equivalência das duas sequencias que representassem estas formulas. Da forma como está, uma mesma formula pode ser equivalente a duas sequencias diferentes, o que geraria uma inconsistência.

conditions shall be a *tautology*. For example, if the engineer defines that a component produces as outputs: (i)  $A$  if  $F_1$  occurs, and (ii)  $B$  if  $F_2$  occurs, then there should be an output for condition  $\neg F_1 \wedge \neg F_2$ , and  $A$  and  $B$  must converge when both  $F_1$  and  $F_2$  occur. By convergence, an initial idea is that  $A = B$  in this case. To connect components and to generate **FTs**, we ask questions to the **AL** formulas: we define *predicates*.

A nominal value is required to handle the conditions that do not result in a failure. Recall from previous example, if  $F_1$  and  $F_2$  do not occur, then the system should be in a normal state, and its output is signalled as nominal with some nominal value. Nominal values are used to analyse value-based failures. In general, failure outputs do not have an associated value.

In some cases a degraded state can be an undesired state, as for example, if one wants to check the probability of operating in high-cost conditions. For these situations, the output values are signalled as degraded, but they have an associated value.

To connect components, instead of using a fixed condition like  $F_1$  and  $F_2$ , we use a predicate. For example: if an omission is detected in the first input, the output of this component is  $A$ ; the component outputs  $B$  if  $F_1$  occurs, and it outputs its nominal value, otherwise.

To obtain a fault tree from **AL** we define a predicate over a whole **AL** formula of a system. For example: what are the conditions that generate and output *omission*?

The underlying conditions in **AL** can be in Boolean algebra or in **ATF**. In any case, soundness and completeness in **AL** is given in terms of the underlying algebra: given a formula in **AL**, any predicate generates a valid expression in the underlying algebra, and there exists a formula and a predicate for any expression in the underlying algebra.

**Proposition 4.2** (Soundness and completeness of **AL**).<sup>5</sup> *Let  $F$  be the set of all formulas in **AL**,  $G$  be the set of all formulas in its underlying algebra, and  $P$  a predicate over output values of **AL**, then:*

$$\forall f \in F. \exists g \in G. P(f) \equiv g \quad \text{Soundness} \quad (4.30a)$$

$$\forall g \in G. \exists f \in F. P(f) \equiv g \quad \text{Completeness} \quad (4.30b)$$

<sup>5</sup> AS Note:  $P$  não deveria estar quantificado universalmente no primeiro predicado? E no segundo, a quantificação é existencial mesmo? A noção de soundness, como no caso anterior, continua parecendo muito fraca.



## 5 Reasoning fault activation



## 6 Case study

1 2

EMBRAER provided us with the Simulink model of an Actuator Control System (depicted in Figure 22). The failure expression of this system (that is, for each of its constituent components) was also provided by EMBRAER (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 [88, 89]. 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 23.

Now we show two contributions: (i) using only Boolean operators, thus ignoring ordering, we can obtain the same results obtained in [28], and (ii) we represent each of the fault traces reported in [28] as a term in our proposed algebra of temporal faults. Similarly to the association of fault events of Table 7 in Section 3.5, we associate the fault events as:

$$\begin{array}{ll}
 a = \text{LowPower-In1} & A = \mathbf{var} \ a \\
 b = \text{LowPower-In2} & B = \mathbf{var} \ b \\
 s = \text{SwitchFailure} & S = \mathbf{var} \ s
 \end{array}$$

### 6.1 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**. For each trace shown in Section 3.5, a mapping function<sup>3</sup> ( $\tilde{B}$ )

<sup>1</sup> AS Note: Vocês confirmaram com a Embraer que isso pode ser divulgado mesmo? No caso de Gustavo, mesmo exemplos que eles descaracterizam, não permitem que apresentemos assim de forma explícita a modelagem. Nós temos apresentado apenas os resultados.

<sup>2</sup> AS Note: Ficou esquisito não utilizar o not no estudo de caso, após tanta ênfase dada ao operador. Poderia tentar estender o estudo de caso de forma a usar o not.

<sup>3</sup> In this work we do not show the mapping function from traces to **ATF** (and the mapping function with **XBefore** in Section 6.2). The mapping rules follow the traces: **XBefore** is obtained by the order of occurrence and the absence of an event is the complement ( $-$ ).

generates the following sets of lists:

TRACE 1:	$[s, b] \overset{\sim}{B} S \cap B \cap -A$	$\{[s, b], [b, s]\}$
TRACE 2:	$[b, s] \overset{\sim}{B} B \cap S \cap -A$	$\{[s, b], [b, s]\}$
TRACE 3:	$[a, b] \overset{\sim}{B} A \cap B \cap -S$	$\{[a, b], [b, a]\}$
TRACE 4:	$[b, a] \overset{\sim}{B} B \cap A \cap -S$	$\{[a, b], [b, a]\}$
TRACE 5:	$[a, s] \overset{\sim}{B} A \cap S \cap -B$	$\{[a, s], [s, a]\}$
TRACE 6:	$[a, s, b] \overset{\sim}{B} A \cap S \cap B$	$\{[a, b, s], [a, s, b], \dots, [s, b, a]\}$
TRACE 7:	$[a, b, s] \overset{\sim}{B} A \cap B \cap S$	$\{[a, b, s], [a, s, b], \dots, [s, b, a]\}$
TRACE 8:	$[b, a, s] \overset{\sim}{B} B \cap A \cap S$	$\{[a, b, s], [a, s, b], \dots, [s, b, a]\}$

They represent the same faults shown in Section 3.5. Note that the negation in the formula is very simple to represent in ATF (and FBA) because it is just the absence of the generator.

Combining the above sets with unions (ORs), we obtain the following formula set:

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

If we use Boolean expression reduction instead, it results in the following expression in ATF (and in FBA):

$$(A \cap B) \cup (S \cap (A \cup B))$$

which is equivalent to the set of sequences above and is equivalent to EMBRAER failure expression shown in Table 7 (with AND gates as  $\cap$  and OR gates as  $\cup$ ). This shows that ATF can represent (static) failure expression as in our previous work [28].

## 6.2 Structure expressions with XBefore

Now, by using ATF with the XBefore operator and a mapping function ( $\overset{\sim}{XB}$ ), we can capture each possible individual sequences as generated by the work [28]:

TRACE 1:	$[s, b] \overset{\sim}{XB} (S \rightarrow B) \cap -A$	$\{[s, b]\}$
TRACE 2:	$[b, s] \overset{\sim}{XB} (B \rightarrow S) \cap -A$	$\{[b, s]\}$
TRACE 3:	$[a, b] \overset{\sim}{XB} (A \rightarrow B) \cap -S$	$\{[a, b]\}$
TRACE 4:	$[b, a] \overset{\sim}{XB} (B \rightarrow A) \cap -S$	$\{[b, a]\}$
TRACE 5:	$[a, s] \overset{\sim}{XB} (A \rightarrow S) \cap -B$	$\{[a, s]\}$
TRACE 6:	$[a, s, b] \overset{\sim}{XB} A \rightarrow S \rightarrow B$	$\{[a, s, b]\}$
TRACE 7:	$[a, b, s] \overset{\sim}{XB} A \rightarrow B \rightarrow S$	$\{[a, b, s]\}$
TRACE 8:	$[b, a, s] \overset{\sim}{XB} B \rightarrow A \rightarrow S$	$\{[b, a, s]\}$



Using *ATF* and combining each trace with *ORs* (unions), we obtain the following set:

$$M_L = \{[a, b], [b, a], [b, s], [s, b], [a, s], [a, b, s], [a, s, b], [s, a, b]\}$$

From the above traces, we also build an *ATF* expression by mapping each trace to an *XBefore* expression, composing all resulting *XBefore* expressions with *ORs* and reducing them using the *XBefore* laws (Section 4.2), resulting in an expression ( $M_A$ ) that is equivalent to the above set of lists ( $M_L \equiv M_A$ ). The failure expression of the monitor<sup>4</sup> is:

$$\begin{aligned}
M_A &= ((S \rightarrow B) \cap \neg A) \cup ((B \rightarrow S) \cap \neg A) \cup \\
&\quad ((A \rightarrow B) \cap \neg S) \cup ((B \rightarrow A) \cap \neg S) \cup \\
&\quad ((A \rightarrow S) \cap \neg B) \cup \\
&\quad (A \rightarrow S \rightarrow B) \cup (A \rightarrow B \rightarrow S) \cup (B \rightarrow A \rightarrow S) \\
&= (B \cap S \cap \neg A) \cup && \text{by (4.15b)} \\
&\quad (B \cap A \cap \neg S) \cup && \text{by (4.15b)} \\
&\quad ((A \rightarrow S) \cap \neg B) \cup \\
&\quad (A \rightarrow S \rightarrow B) \cup (A \rightarrow B \rightarrow S) \cup (B \rightarrow A \rightarrow S) \\
&= (B \cap S \cap \neg A) \cup \\
&\quad (B \cap A \cap \neg S) \cup \\
&\quad ((A \rightarrow S) \cap \neg B) \cup \\
&\quad ((A \rightarrow S) \cap B) && \text{by (4.17)} \\
&= (B \cap S \cap \neg A) \cup (B \cap A \cap \neg S) \cup (A \rightarrow S) && \text{by absorption}
\end{aligned}$$

The semantics of the above expression is: (i) fault  $b$  (**var**  $b$ ) occurs and fault  $a$  (**var**  $a$ ) or fault  $s$  (**var**  $s$ ) occurs (but not both  $a$  and  $s$ ), or (ii) fault  $a$  occurs before fault  $s$ , which is more precise than the expression found without considering order of events.

<sup>4</sup> In the final formula,  $(B \cap S \cap \neg A) \cup (A \cap B \cap \neg S)$  is equivalent to  $(B \cap (S \oplus A))$ . There is a typo in our previous work [86]. The expression was written with an *OR* ( $\vee$ ) but it should be an *XOR* ( $\oplus$ ).



## 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, 90]. Furthermore, we still connect the strategy presented in [91] with the works reported in [31] (functional analysis) and in [90, 25] (safety assessment) because our new algebra is at least a Boolean algebra.

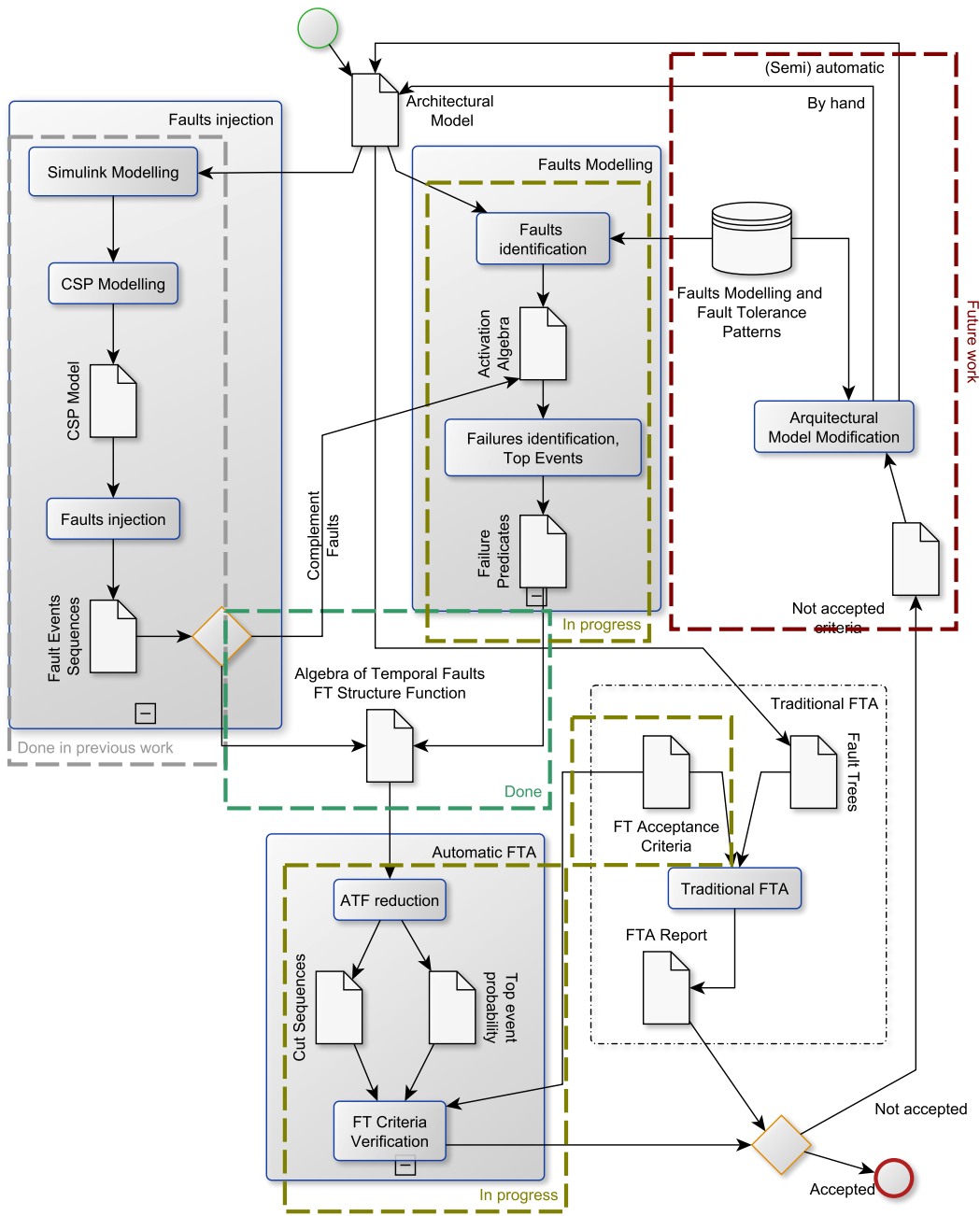
The work reported in [21, 20, 33] tackles simultaneity with “nearly simultaneous” events [92]. But we consider instantaneous events, like the work reported in [23], because we assume that simultaneity is probabilistically impossible.

### 7.1 Status

In Figure 25 we show: (i) what was done in previous work and is used as input, (ii) what was done in the current work, (iii) what will be done in the next months (see Table 8 for tasks schedule), and (iv) what will be done in future work, after the thesis’ defence. Many of the tasks shown in Table 8 already started. The Table shows the estimated execution of the tasks by year’s quarters. The second quarter of 2016 is from April to June/2016, the third quarter is from July to September/2016, the fourth quarter, from October to December/2016, and we expected to defend the thesis by February/2017.

**Table 8** – Tasks schedule

Task	2nd	3rd	4th	1st
Qualification	•			
Elaborate a theory for the AL	•			
Elaborate a theory for the acceptance criteria	•			
Prepare a paper about AL and acceptance criteria	•	•		
Submit paper about AL and acceptance criteria		•		
Prove soundness and completeness theorems for the DNF of ATF		•	•	
Define the mapping rules from traces to ATF			•	
Demonstrate the relations of NOT and XBefore and other operators			•	
Define the conditions that cause non-coherent analysis with NOT			•	
Write the results in the thesis		•	•	
Prepare thesis’ defence			•	•
Defence				•



**Figure 25** – Status of this thesis using the strategy overview (see Figure 1)

## 7.2 Next steps in this thesis

The next steps in this thesis are the conclusion of the work-in-progress of the “Fault Modelling” and “Automatic FTA” blocks in Figure 25. An initial version of the theory of the **AL** is done. The case study shown in this thesis is also modelled in **AL**. But we need to adapt the failure predicates to **ATF**. It may require a full refactoring of the theory of **AL**.

We developed a small set of rules for the acceptance criteria verification, but we need to add more sophisticated rules, as for example, to consider phase and latency.

We showed the **DNF** for **ATF**, but we did not demonstrate that every formula

can be converted into **DNF**. The laws shown in this work should be sufficient for this demonstration. Also, we did not show the mapping rules from traces to **ATF** (with Boolean operators only **AND** with **XBefore**). The mapping rules follow those for the traces: **XBefore** is obtained by the order of occurrence and the absence of an event is the complement ( $-$ ).

Although we do not use negation (**NOT** operator) with **XBefore** in our case study, it is part of **ATF**, so it could be used. As future work we will demonstrate the relations of **NOT** and **XBefore**, as we did for **AND** and **OR**. We will also define laws to avoid the conditions that cause non-coherent analysis [11]. The issue with negated events comes up when both an event and its negation appear on the same tree. One very restrictive solution to this issue is applying the *generators independence* laws (4.8c, 4.8e) on basic events of a tree, by actually considering a new event  $ne$  in place of the negation of another event  $e$  (for instance,  $\text{var } e$  and  $\text{var } ne = - \text{var } e$ ). We look forward to obtain a less restrictive law.

### 7.3 Future work, out of the scope of this thesis

Boolean formulas reduction can be achieved by: (i) application of Boolean laws, (ii) **BDD**, or (iii) **FBA**s. 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 **DTMC**s (**Markov chain**) is more appropriate to represent several states than **SFT**s. Considering that **DFT**s were conceived as a visual representation of **Markov chains**, then we may say that **DFT**s can be used to represent several states. Thus they are suitable to propose the architectural model modifications as shown in Figures 1 and 25. 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](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: <http://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](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](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](http://dx.doi.org/10.1016/S1567-8326(02)00067-X).
- 41 BAIER, C.; HAVERKORT, B.; HERMANN, 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](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](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](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](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.; DERAÏN, 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](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 O'CONNOR, P.; NEWTON, D.; BROMLEY, R. *Practical reliability engineering*. [S.l.]: Wiley, 2002. ISBN 9780470844632.
- 89 KOREN, I.; KRISHNA, C. M. *Fault Tolerant Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN 0120885255.
- 90 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.



- 91 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.
- 92 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.
- 93 HAFTMANN, F.; LOCHBIHLER, A. *Dlist theory*. Available from Internet: [http://isabelle.in.tum.de/library/HOL/HOL-Quickcheck\\_Examples/Dlist.html](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 [93] (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 ??.

## 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* ("( $3_{\_ \dots}$ )" [80,80,80] 80)

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

**definition** *slice\_right* :: "'a::sliceable  $\Rightarrow$  nat  $\Rightarrow$  'a" ("( $2_{\_ \dagger \dots}$ )" [80,80] 80)  
**where** "*slice\_right* x i =  $x_{\dagger 0..i}$ "

**notation** ("latex") *slice\_right* ("( $2_{\_ \dots}$ )" [80,80] 80)

**definition** *slice\_left* :: "'a::sliceable  $\Rightarrow$  nat  $\Rightarrow$  'a" ("( $2_{\_ \dagger \dots}$ )" [80,80] 80)  
**where** " $x_{\dagger i..} = x_{\dagger i..}(\# x)$ "

**notation** ("latex") *slice\_left* ("( $2_{\_ \dots}$ )" [80,80] 80)

### A.1.1 Disjoint elements and sliceable

**lemma** (in *sliceable*) *slice\_right\_disjoint[simp]*: "*disjoint* xs  $\Rightarrow$   
*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  $\Rightarrow$   
*disjoint* ( $xs_{\dagger i..}$ )"  
**unfolding** *slice\_left\_def*  
**by** *simp*

### A.1.2 n-th element in a sliceable

**abbreviation** *sliceable\_nth* :: "'a::sliceable  $\Rightarrow$  nat  $\Rightarrow$  'a"  
**where**  
"*sliceable\_nth* l i  $\equiv$   $l_{\dagger i..}(\text{Suc } i)$ "

### A.1.3 Theorems for sliceable

**theorem** (in *sliceable*) *empty\_seq\_inter\_eq [simp]*:  
"*disjoint* x  $\Rightarrow$  *empty\_inter* ( $x_{\dagger ..i}$ ) ( $x_{\dagger i..}$ )"  
**by** (*simp add: slice\_right\_def slice\_left\_def*)

**theorem** (in *sliceable*) *empty\_seq\_sliced\_inter [simp]*:

```

"disjoint x  $\implies$  b  $\leq$  i  $\implies$  j  $\leq$  a  $\implies$  i  $\leq$  j  $\implies$  a  $\leq$  size x  $\implies$ 
  empty_inter (x†b..i) (x†j..a)"
proof-
  let ?l = "x†b..a"
  assume lt0: "i  $\leq$  j"
  assume lt1: "j  $\leq$  a"
  assume lt2: "b  $\leq$  i"
  assume lt3: "a  $\leq$  size x"
  assume lt4: "disjoint x"
  have blta: "b  $\leq$  a" using lt0 lt1 lt2 by simp
  have ilta: "i  $\leq$  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  $\implies$  i  $\leq$  j  $\implies$ 
    set (take i (drop 0 l))
       $\cap$  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).."
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)..

```

```

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:

" $\forall i j . (\text{disjoint } x \wedge i \neq j \wedge i < (\#x) \wedge j < (\#x)) \longrightarrow$   
 $(x \upharpoonright i..(\text{Suc } i) \neq x \upharpoonright j..(\text{Suc } j))$ "

by (simp add: local.disjoint\_slice\_suc)

lemma (in sliceable) empty\_slice\_none:

" $(\#x) = 0 \implies (\#(x \upharpoonright i..j)) = 0$ "

by (simp add: size\_slice)

corollary (in sliceable) empty\_slice\_right\_none:

" $(\#x) = 0 \implies (\#(x \upharpoonright ..j)) = 0$ "

by (simp add: slice\_right\_def sliceable\_class.empty\_slice\_none)

corollary (in sliceable) empty\_slice\_left\_none:

" $(\#x) = 0 \implies (\#(x \upharpoonright i..)) = 0$ "

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

" $l \upharpoonright i..f = \text{Dlist } (\text{take } (\text{max } 0 (f-i)) (\text{drop } i (\text{list\_of\_dlist } l)))$ "

definition

" $\text{size } l = \text{length } (\text{list\_of\_dlist } l)$ "

definition

"empty\_inter l k =  
 $((\text{set } (\text{list\_of\_dlist } l)) \cap (\text{set } (\text{list\_of\_dlist } k)) = \{\})$ "

definition

"disjoint l = distinct (list\_of\_dlist l)"

lemma list\_of\_dlist\_slice :

" $\text{list\_of\_dlist } (l \upharpoonright i..f) = \text{take } (\text{max } 0 (f-i)) (\text{drop } i (\text{list\_of\_dlist } l))$ "

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: " $c \leq k \implies$

(  
set (take c (list\_of\_dlist x))  $\cap$   
set (drop k (list\_of\_dlist x))  
) = {}"

by (simp add: set\_take\_disj\_set\_drop\_if\_distinct)

lemma Dlist\_forall\_slice\_eq1:

"( $\forall i f. (Dlist (take (max 0 (f-i)) (drop i (list_of_dlist l1))) =$   
Dlist (take (max 0 (f-i)) (drop i (list\_of\_dlist l2))))  $\implies$   
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  $\longleftrightarrow$   
( $\forall i f. (Dlist (take (max 0 (f-i)) (drop i (list_of_dlist l1))) =$   
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  $\implies i \neq j \implies i < \text{length } l \implies j < \text{length } l \implies$   
take 1 (drop i l)  $\neq$  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  $\dagger$  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_OL
        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_OL
        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_OL)
  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)  $\implies$ 
  (Dlist (x # xs)) $\dagger$ .. $(\text{Suc } n) = \text{Dlist } (x \# (\text{list\_of\_dlist } ((\text{Dlist } xs)\dagger..n)))$ "
unfolding slice_right_def slice_dlist_def
by (simp add: distinct_remdups_id)

```

```

theorem slice_append:
  " $\forall n. \text{Dlist } ((\text{list\_of\_dlist } (l\dagger..n)) @ (\text{list\_of\_dlist } (l\dagger n..))) = l$ "
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:
  " $\forall i \ s \ e. s \leq i \wedge i \leq e \implies$ 
   $((\text{list\_of\_dlist } (l\dagger s..i)) @ (\text{list\_of\_dlist } (l\dagger i..e))) =$ 
   $\text{list\_of\_dlist } (l\dagger s..e)$ "
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:
  " $\forall i \ j. i \leq j \implies$ 
   $((\text{list\_of\_dlist } (l\dagger..i)) @$ 
   $(\text{list\_of\_dlist } (l\dagger i..j)) @ (\text{list\_of\_dlist } (l\dagger j..))) = \text{list\_of\_dlist } l$ "
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]:
  " $i \leq j \implies \text{set } (\text{list\_of\_dlist } (l\dagger..i)) \cap \text{set } (\text{list\_of\_dlist } (l\dagger j..)) = \{\}$ "
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]:
  " $\text{set } (\text{list\_of\_dlist } (l\dagger..i)) \cap \text{set } (\text{list\_of\_dlist } (l\dagger i..)) = \{\}$ "
by simp

```

```

corollary distinct_slice_lt_inter_empty [simp]:
  " $i < j \implies \text{set } (\text{list\_of\_dlist } (l\dagger..i)) \cap \text{set } (\text{list\_of\_dlist } (l\dagger j..)) = \{\}$ "
by simp

```

```

corollary distinct_slice_diff1:
  " $\text{set } (\text{list\_of\_dlist } (l\dagger..i)) - \text{set } (\text{list\_of\_dlist } (l\dagger i..)) =$ 
   $\text{set } (\text{list\_of\_dlist } (l\dagger..i))$ "

```

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  $\implies$  take (Suc 0) l = [hd l]"
```

```
apply (induct l)
```

```
by auto
```

```
lemma take_drop_suc: "i < length l  $\implies$  length l > 0  $\implies$ 
```

```
  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..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  $\neq$  []  $\implies$  take (Suc 0) l = [hd l]"
```

```
apply (induct l, simp)
```

```
by auto
```

```
lemma take_one_singleton_nth: "l  $\neq$  []  $\implies$  take (Suc 0) l = [l!0]"
```

```
apply (induct l, simp)
```

```
by auto
```

```
lemma take_one_drop_n_append_singleton_nth:
```

```
  "ys  $\neq$  []  $\implies$  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  $\neq$  []  $\implies$  [(xs @ ys)!(length xs)] = [hd ys]"
```

```
by (induct ys, auto)
```

```
lemma take_one_drop_n_singleton_nth: "l  $\neq$  []  $\implies$  n < length l  $\implies$ 
```

```
  take 1 (drop n l) = [l!n]"
```

```
proof-
```

```
  assume 0: "l  $\neq$  []"
```

```
  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:

" $i \leq j \implies \text{list\_of\_dlist } (l \upharpoonright i..) =$   
 $(\text{list\_of\_dlist } (l \upharpoonright i..j)) @ (\text{list\_of\_dlist } (l \upharpoonright j..))$ "

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\_OL size\_dlist\_def take\_all)

lemma dlist\_disjoint[simp]: "disjoint (l::'a dlist)"

by (simp add: disjoint\_dlist\_def)

lemma dlist\_member\_suc\_nth1:

" $x \in \text{set } (\text{list\_of\_dlist } (l \upharpoonright i..(\text{Suc } i))) \implies x = (\text{list\_of\_dlist } l)!i$ "

proof-

assume 0: " $x \in \text{set } (\text{list\_of\_dlist } (l \upharpoonright i..(\text{Suc } i)))$ "

obtain r1 where 1: " $r1 = \text{list\_of\_dlist } l$ " by blast

hence " $x \in \text{set } (\text{take } (\text{max } 0 (\text{Suc } i - i)) (\text{drop } i \ r1))$ "

using 0 by (metis list\_of\_dlist\_slice )

hence " $x \in \text{set } (\text{take } 1 (\text{drop } i \ r1))$ " by simp

hence " $x = r1!i$ "

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:

" $i < (\#l) \implies x = (\text{list\_of\_dlist } l)!i \implies$

$x \in \text{set } (\text{list\_of\_dlist } (l \upharpoonright i..(\text{Suc } i)))$ "

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: " $i < (\#l) \implies$

$(x = (\text{list\_of\_dlist } l)!i) \longleftrightarrow (x \in \text{set } (\text{list\_of\_dlist } (l \upharpoonright i..(\text{Suc } i))))$ "

using dlist\_member\_suc\_nth1 dlist\_member\_suc\_nth2

by fastforce

corollary not\_dlist\_member\_empty[simp]:

" $\neg \text{Dlist.member } (\text{Dlist.empty}) \ v$ "

" $\neg (\text{Dlist.member } (\text{Dlist } []) \ v)$ "

by (simp add: Dlist.member\_def Dlist.empty\_def List.member\_def)+

lemma dlist\_empty\_slice\_none: " $(\text{Dlist.empty} \upharpoonright i..j) = \text{Dlist.empty}$ "

```
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_takenD 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:
  "(\exists i. sliceable_nth dl i = Dlist [v]) \longleftrightarrow 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 tempo1

```

class temporal_faults_algebra_basic = boolean_algebra +
  fixes neutral :: "'a"
  fixes xbefore :: "'a \Rightarrow 'a \Rightarrow 'a"
  fixes tempo1 :: "'a \Rightarrow bool"
  assumes xbefore_bot_1: "xbefore bot a = bot"
  assumes xbefore_bot_2: "xbefore a bot = bot"
  assumes xbefore_neutral_1: "tempo1 a \Longrightarrow xbefore neutral a = a"
  assumes xbefore_neutral_2: "tempo1 a \Longrightarrow xbefore a neutral = a"
  assumes xbefore_not_idempotent: "tempo1 a \Longrightarrow xbefore a a = bot"
  assumes inf_tempo1: "[tempo1 a; tempo1 b] \Longrightarrow tempo1 (inf a b)"
  assumes xbefore_not_sym:
    "[tempo1 a; tempo1 b] \Longrightarrow (xbefore a b) \leq -(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] \Longrightarrow
    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 \Rightarrow 'a \Rightarrow bool"
  fixes tempo2 :: "'a \Rightarrow bool"
  fixes tempo3 :: "'a \Rightarrow bool"
  fixes tempo4 :: "'a \Rightarrow 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  $\implies$   $\llbracket$ tempo1 a; tempo1 b $\rrbracket \implies$   
 $\llbracket$ tempo2 a; tempo2 b $\rrbracket \implies \llbracket$ tempo3 a; tempo3 b $\rrbracket \implies \llbracket$ tempo4 a; tempo4 b $\rrbracket \implies$   
 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  $\implies$   $\llbracket$ tempo1 b; tempo1 c $\rrbracket \implies$   
 $\llbracket$ tempo2 b; tempo2 c $\rrbracket \implies \llbracket$ tempo3 b; tempo3 c $\rrbracket \implies \llbracket$ tempo4 b; tempo4 c $\rrbracket \implies$   
 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  $\implies$   $\llbracket$ tempo1 a; tempo1 b $\rrbracket \implies$   
 $\llbracket$ tempo2 a; tempo2 b $\rrbracket \implies \llbracket$ tempo3 a; tempo3 b $\rrbracket \implies \llbracket$ tempo4 a; tempo4 b $\rrbracket \implies$   
 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  $\implies$   $\llbracket$ tempo1 a; tempo1 b $\rrbracket \implies$

$\llbracket \text{tempo2 } a; \text{ tempo2 } b \rrbracket \implies \llbracket \text{tempo3 } a; \text{ tempo3 } b \rrbracket \implies \llbracket \text{tempo4 } a; \text{ tempo4 } b \rrbracket \implies$   
 $\text{sup } a \text{ (xbefore } a \text{ } b) = a$   
 by (metis dual\_order.trans inf.cobounded1 sup.absorb1 sup.cobounded1  
 xbefore\_sup\_equiv\_inf)

corollary xbefore\_sup\_absorb\_1:

$\text{"independent\_events } a \text{ } b \implies \llbracket \text{tempo1 } a; \text{ tempo1 } b \rrbracket \implies$   
 $\llbracket \text{tempo2 } a; \text{ tempo2 } b \rrbracket \implies \llbracket \text{tempo3 } a; \text{ tempo3 } b \rrbracket \implies \llbracket \text{tempo4 } a; \text{ tempo4 } b \rrbracket \implies$   
 $\text{sup (xbefore } a \text{ } b) \text{ } a = a$

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:

$\text{"independent\_events } a \text{ } b \implies \llbracket \text{tempo1 } a; \text{ tempo1 } b \rrbracket \implies$   
 $\llbracket \text{tempo2 } a; \text{ tempo2 } b \rrbracket \implies \llbracket \text{tempo3 } a; \text{ tempo3 } b \rrbracket \implies \llbracket \text{tempo4 } a; \text{ tempo4 } b \rrbracket \implies$   
 $\text{sup } a \text{ (xbefore } b \text{ } a) = a$

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  $\implies$

$\llbracket \text{tempo1 } a; \text{ tempo1 } b \rrbracket \implies$   
 $\llbracket \text{tempo2 } a; \text{ tempo2 } b \rrbracket \implies$   
 $\llbracket \text{tempo3 } a; \text{ tempo3 } b \rrbracket \implies$   
 $\llbracket \text{tempo4 } a; \text{ tempo4 } b \rrbracket \implies$   
 $\text{inf } a \text{ (xbefore } a \text{ } b) = \text{xbefore } a \text{ } b$

by (simp add: local.inf\_absorb2 local.le\_iff\_sup xbefore\_sup\_absorb\_1)

theorem xbefore\_inf\_absorb\_2: "independent\_events a b  $\implies$

$\llbracket \text{tempo1 } a; \text{ tempo1 } b \rrbracket \implies$   
 $\llbracket \text{tempo2 } a; \text{ tempo2 } b \rrbracket \implies$   
 $\llbracket \text{tempo3 } a; \text{ tempo3 } b \rrbracket \implies$   
 $\llbracket \text{tempo4 } a; \text{ tempo4 } b \rrbracket \implies$

```

inf a (xbefore b a) = xbefore b a"
by (simp add: local.inf.absorb2 local.sup.absorb_iff1 xbefore_sup_absorb_2b)

```

corollary inf\_xbefore\_equiv\_sups\_xbefore\_expanded:

```

"independent_events a b  $\implies$  independent_events a c  $\implies$ 
 $\llbracket$ tempo1 a; tempo1 b; tempo1 c $\rrbracket \implies \llbracket$ tempo2 a; tempo2 b; tempo2 c $\rrbracket \implies$ 
 $\llbracket$ tempo3 a; tempo3 b; tempo3 c $\rrbracket \implies \llbracket$ tempo4 a; tempo4 b; tempo4 c $\rrbracket \implies$ 
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

lemma xbefore\_sup\_compl\_inf\_absorb1:

```

"independent_events a b  $\implies \llbracket$ tempo1 a; tempo1 b $\rrbracket \implies$ 
 $\llbracket$ tempo2 a; tempo2 b $\rrbracket \implies \llbracket$ tempo3 a; tempo3 b $\rrbracket \implies \llbracket$ tempo4 a; tempo4 b $\rrbracket \implies$ 
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: "- sup (inf a (- b)) (xbefore a b) = inf (sup (- a) (- (- b)))
(sup (sup (- a) (- b)) (xbefore b a))"
  using a8 a7 a6 a5 a4 a3 a2 a1 local.compl_inf local.compl_sup local.not_xbefore
by presburger
have f11: "sup (- a) (- a) = - a"
  by auto
have f12: "∀f a aa. ¬ abel_semigroup f ∨ f (a::'a) aa = f aa a"
  by (simp add: abel_semigroup commute)
then have f13: "inf (sup (- a) (- a)) (xbefore b a) = inf (xbefore b a) (- a)"
  using f11 local.inf.abel_semigroup_axioms by presburger
have f14: "inf b (xbefore b a) = inf (xbefore b a) b"
  using f12 local.inf.abel_semigroup_axioms by blast
have f15: "inf a (xbefore b a) = xbefore b a"
  using a9 a8 a7 a6 a5 a4 a3 a2 a1 by (meson xbefore_inf_absorb_2)
then have f16: "inf (xbefore b a) a = xbefore b a"
  using f12 local.inf.abel_semigroup_axioms by presburger
then have "inf b (xbefore b a) = inf (xbefore b a) (sup (xbefore a b) (xbefore
b a))"
  using f14 a9 a8 a7 a6 a5 a4 a3 a2 a1 by (metis (full_types) local.inf.assoc
local.xbefore_sup_equiv_inf)
then have f17: "inf b (xbefore b a) = inf (sup (xbefore a b) a) (xbefore b a)"
  using f15 a3 a2 by (simp add: local.inf_sup_distrib1 local.inf_sup_distrib2
local.xbefore_inf_equiv_bot)
have f18: "sup (xbefore a b) a = a"
  using a9 a8 a7 a6 a5 a4 a3 a2 a1 by (metis (no_types) xbefore_sup_absorb_1)
have "inf (xbefore b a) (- a) = bot"
  by (metis f11 f13 f16 local.compl_inf_bot local.inf.left_commute local.inf_bot_right)
then have f19: "inf (sup (- a) (- (- b))) (xbefore b a) = inf (sup (xbefore a
b) a) (xbefore b a)"
  using f17 f13 by (simp add: local.inf_sup_distrib2)
have "inf (sup (- a) (- (- b))) (sup (- a) (- b)) = sup (sup (- a) (- a)) (inf
b (- b))"
  using local.distrib_imp1 local.inf_sup_distrib1 by force
then have "- sup (inf a (- b)) (xbefore a b) = - inf (- xbefore b a) a"
  using f10 f15 f18 f19 local.inf_compl_bot local.inf_sup_distrib1 local.sup_commute
by auto
then show ?thesis
  using f12 by (metis (no_types) local.compl_eq_compl_iff local.inf.abel_semigroup_axioms)
qed

```

**end**

[1](#) [2](#) [3](#)

---

<sup>1</sup> AD Note: Verify indexes again (search again for words already in the index)

<sup>2</sup> AD Note: Add more indexes

<sup>3</sup> AD Note: Run “makeindex main” on this directory

# TODOs

- De Augusto: “Este capítulo é desnecessariamente conciso, principalmente por se tratar da contribuição. Seria importante, na versão final, ilustrar cada conceito e propriedade com exemplos. Tem que destacar resultados importantes, como o fato da mecanização com Isabelle. Deveria ter uma seção ou um capítulo dedicado à mecanização, não incluindo tudo, obviamente, mas ilustrando um pequeno exemplo.” . . . . . 71
- Explain the interaction of (i) fault-injection, (ii) theorem proving, and (iii) symbolic execution. . . . . 71
- dizer que nossa abordagem usa a abordagem similar à de expressões de estrutura, mas que tem o objetivo de dar uma semântica denotacional baseada em conjuntos 71
- Verificar significado de altogether . . . . . 73