



Aralia User Manual



Aralia User Manual

Copyrights (c) 2009 DASSAULT SYSTEMS
DASSAULT SYSTEMS
10, rue Marcel Dassault
78946 Vélizy Villacoublay Cedex
FRANCE
www.3ds.com

Date: November 02, 2009
Version: 4.8a

Table of Contents

I	Introduction.....	7
I.1	What is Aralia	7
I.2	Notational conventions.....	7
I.3	How to read this manual	8
II	Boolean Formulae	9
II.1	Equations	9
II.2	Connectives	9
II.3	Shannon Decomposition	10
II.4	Quantifiers.....	10
II.5	Concrete Syntax for Boolean Formulae	11
II.6	Terminology	12
II.7	Summary.....	12
III	Data Structures	13
III.1	Binary Decision Diagrams.....	14
III.2	Zero-Suppressed Binary Decision Diagrams	15
III.3	Sums of Products.....	16
III.4	Handles.....	16
III.5	Bibliography	16
IV	Minimal Cutsets and Prime Implicants.....	18
IV.1	Preliminary Definitions	18
IV.2	Prime implicants and minimal cutsets.....	20
IV.2.1	Prime implicants.....	20
IV.2.2	Minimal cutsets	20
IV.2.3	What do minimal cutsets characterize?.....	22
IV.2.4	Decomposition Theorems	23
IV.3	Commands to compute Prime Implicants and Minimal Cutsets.....	24
IV.4	Summary	26
V	Handling Very Large Models	27
V.1	Tuning BDD Tables	27
V.1.1	BDD indices and the BDD entry table	27
V.1.2	The BDD Unique-table	28
V.1.3	The BDD-Hashtables	28
V.1.4	The BDD-Hashcache	29
VI	Stochastic Layer	30
VI.1	Introduction.....	30
VI.2	Commands	31
VI.3	Parameters	32
VI.3.1	Constant Parameters	32
VI.3.2	Named parameters	32
VI.3.3	Arithmetic operations on parameters	32
VI.3.4	Boolean Operations on Parameters	33
VI.3.5	Conditional Operations on Parameters	33
VI.4	Time-Dependent Distributions	34
VI.4.1	Mission Time	34
VI.4.2	Constant Distribution.....	34

VI.4.3 Exponential Distribution	34
VI.4.4 GLM Distribution	35
VI.4.5 MTT Distribution.....	35
VI.4.6 Weibull Distribution	35
VI.4.7 Dirac Distribution.....	36
VI.4.8 Distributions for Periodically Tested Components.....	36
VI.4.9 Dormant Distribution	38
VI.4.10 Standby Distribution	38
VI.4.11 Bound Time Distribution	40
VI.5 Random Deviates	40
VI.5.1 Uniform Deviates.....	40
VI.5.2 Normal Deviates.....	41
VI.5.3 Lognormal Deviates	41
VI.5.4 Histograms	42
VI.6 Summary	43
VII Probability and Importance Factors	45
VII.1 Probabilities of Gates.....	45
VII.2 Positive Probabilities of Gates	46
VII.3 Conditional Probabilities	46
VII.4 Marginal Importance Factor	47
VII.5 Critical Importance Factor.....	48
VII.6 Diagnostic Importance Factor	48
VII.7 Risk Achievement Worth	48
VII.8 Risk Reduction Worth	49
VII.9 Aralia Commands	49
VII.10 Summary	51
VIII Time Dependent Analyses.....	52
VIII.1 Mathematical Definitions of System Reliability	52
VIII.2 Mathematical Definitions of System Availability	53
VIII.3 Approximations of the Reliability.....	57
VIII.3.1 Murchland Lower Bound	57
VIII.3.2 Barlow-Proschan lower bound	57
VIII.3.3 Vesely Approximations.....	57
VIII.3.4 Equivalent Lambda	58
VIII.4 Aralia Commands	58
VIII.5 Safety Integrity Levels	59
VIII.6 Summary	60
IX Sensitivity Analyses.....	61
IX.1 Why to perform sensitivity analyses?.....	61
IX.2 How to perform sensivity analyses?	61
IX.3 The pseudo-random numbers generator	63
X Selectors	65
X.1 Basic selectors.....	65
X.2 Set operations	65
X.3 Selectors that operate through definitions	66
X.4 Predicate selectors	66
X.5 Sorts	67
X.6 Pattern-matching	67

X.7 Summary	68
XI Expressions, Fields and Attributes	70
XI.1 Expressions	70
XI.2 Fields	70
XI.3 Attributes	71
XII Filters	72
XII.1 What are filters used for?	72
XII.2 Syntax of filters	73
XII.3 Summary	74
XIII Options	75
XIII.1 The command option	75
XIII.2 Available options	75
XIII.2.1 Options for sensitivity analyses	75
XIII.2.2 Options for multiple mission times computations	76
XIII.2.3 Other options	77
XIV Glossary	78
XV Summary of Aralia Commands	79
XV.1 Boolean Equations	79
XV.2 Probability distributions	80
XV.3 Command approximate	81
XV.4 Command attribute	81
XV.5 Command basic-event	81
XV.6 Command bdd-order	81
XV.7 Command clear	82
XV.8 Command coalesce	82
XV.9 Command compute	82
XV.10 Command display	83
XV.11 Command echo	84
XV.12 Command exit	84
XV.13 Command group	84
XV.14 Command help	84
XV.15 Command history	84
XV.16 Command load	84
XV.17 Command normalize	84
XV.18 Command option	85
XV.19 Command parameter	85
XV.20 Command prune	85
XV.21 Command remove	85
XV.22 Command rename	85
XV.23 Command rename	85
XV.24 Command rewrite	85
XV.25 Command set	86
XV.26 Command sop-order	86
XV.27 Command sort	86
XV.28 Command store	86
XV.29 Command store-order	87
XV.30 Command system	87
XV.31 Command timer	87

XV.32 Command trace.....	87
XV.33 Command user-order	87
XV.34 Instructions	88
XV.35 Selectors	88
XV.36 Filters	88
XV.37 Expressions, Attributes, Fields	89

I INTRODUCTION

I.1 What is Aralia

Aralia is a Binary Decision Diagrams engine dedicated to the quantification of Boolean risk assessment models: Fault Trees, Event Trees, Block Diagrams... Aralia user's interface consists in a command interpreter. Aralia is designed to be embedded into workbenches that provide the user with their own Graphical User Interface.

This document describes the command of Aralia (version 4.8). Mathematical concepts are introduced when required. Algorithms are just sketched. Bibliographic references are given for a full exposition of internal Aralia mechanisms.

I.2 Notational conventions

Throughout this document, we use the following notational conventions. Aralia commands are written using *this font*, e.g.

```
compute BDD g1;
```

Non terminal symbols are written in *italic* surrounded with *<* and *>*, e.g.

```
compute BDD <variable-selector> ;
```

Optional parts of commands are surrounded with *[* and *]*, e.g.

```
display <variable-selector> [ <redirection> ] ;
```

To represent *n* consecutive occurrences of a construct *c* separated with a given separator *s*, we use the notation '*[c s] n*', e.g in the following command, the Weibull distribution takes 3 arguments (parameters).

```
basic-event set <selector> Weibull( [ <parameter> , ] 3 );
```

Therefore the following constructs are equivalent.

```
Weibull( <parameter>, <parameter>, <parameter> );  
Weibull( [ <parameter> , ] 3 );
```

I.3 How to read this manual

This manual is organized as follows.

- Chapter II describes Boolean formulae and their concrete syntax in Aralia.
- Chapter III describes Aralia data structures.
- Chapter IV introduces the notions of minimal cutsets and prime implicants as well as the commands to compute them.
- Chapter V presents how to tune the Aralia engine to handle large models.
- Chapter VI gives the available probability distributions for basic events.
- Chapter VII presents the commands to compute probabilities and importance factors.
- Chapter VIII presents the commands to compute approximations of the reliability of the system under study.
- Chapter IX describes how to perform sensitivity analyses.
- Chapter X presents selectors.
- Chapter XI presents expressions, fields and attributes.
- Chapter XII presents filters.
- Chapter XIII presents options set through the command “option”.
- A glossary of terms is given chapter XIV.
- A summary of Aralia commands is given chapter XV.
- Finally, (non exhaustive) list of changes between versions is given chapter XVI.

II BOOLEAN FORMULAE

II.1 Equations

Aralia maintains a set of Boolean equations of the form $v = F$, where v is a Boolean variable and F is a Boolean formula. F may depend on variables that appear themselves as the left member of an equation, and so on. A variable should occur at most once as the left member of an equation and the set of equations should not contain loops, i.e. that if a variable v depends eventually on another variable w , then w cannot depend on v .

The set of equations is called the store in the Aralia terminology. It describes a combinatorial circuit. The inputs of this circuit are the variables that do not occur as the left member of an equation. The outputs of the circuit are the variables that do not occur in the right member of an equation.

From a reliability engineering point of view, outputs represent top events of fault trees (or more generally undesirable events) while inputs represent terminal (or basic) events. Note that several trees may coexist within the same store, possibly sharing subsystems.

II.2 Connectives

Boolean formulae are built over the variables, the two constants 0 (false) and 1 (true) and the following connectives.

- Disjunction, denoted in the sequel by \vee or $+$. E.g. $a \vee b$, $a+b$.
- Conjunction, denoted in the sequel by \wedge or $'.'$ or even omitted. E.g. ab , $a.b$, $a \wedge b$.
- Negation, denoted in the sequel by \neg or $-$. E.g., $\neg a$, $-a$.
- Implication, denoted in the sequel \Rightarrow E.g. $a \Rightarrow b$
- If-and-only-if, denoted in the sequel by \Leftrightarrow .
- Exclusive or, denoted in the sequel by \oplus .

Notations are changed when it is convenient. Usually, variables are denoted by letters: a , b , v ... and formulae (or gates) are denoted by capital letters F , G ...

\vee , \wedge , \Leftrightarrow and \oplus are associative and commutative. The truth table for these connectives is as follows.

F	G	$\neg F$	$\neg G$	$F.G$	$F+G$	$F\Rightarrow G$	$F\Leftarrow G$	$F\oplus G$
1	1	0	0	1	1	1	1	0
1	0	0	1	0	1	0	0	1
0	1	1	0	0	1	1	0	1
0	0	1	1	0	0	1	1	0

Aralia makes also the following connectives available.

- If-Then-Else, denoted by $F \rightarrow G, H$. Let F, G and H be three Boolean formulae, then $F \rightarrow G, H = F.G + \neg F.H$.
- k-out-of-n, denoted by $@(k, [F_1, \dots, F_n])$, where k is an integer such that $0 \leq k \leq n$. $@(k, [F_1, \dots, F_n])$ is true when at least k out of the F_i 's are true.
- Cardinality operator, denoted by $\#(l, h, [F_1, \dots, F_n])$, where l and h are two integers such that $0 \leq l \leq h \leq n$. $\#(l, h, [F_1, \dots, F_n])$ is true when at least l and at most h out of the F_i 's are true.

II.3 Shannon Decomposition

The If-Then-Else connective plays a central role in Binary Decision Diagrams, and therefore in the whole Aralia mathematical framework. This connective together with the two constants 0 and 1 form a complete basis for the Boolean algebra, i.e. that any formula can be rewritten using only variables, 0, 1 and If-Then-Else's. To achieve this goal, one uses the Shannon decomposition.

Shannon Decomposition: Let F be a formula and v a variable that occurs in F . Then there exist two formulae F_0 and F_1 , not depending on v , such that $F = v \rightarrow F_1, F_0$.

The process can be reiterated on F_0 and F_1 . F_0 and F_1 are obtained by substituting respectively the constants 0 and 1 for the variable v in F . In that case, F_0 and F_1 are also denoted by $F[0/v]$ and $F[1/v]$ or by $F_{\neg v}$ and F_v .

$F_{\neg v}$ and F_v are called the (positive and negative) cofactors of F w.r.t. v . By extension, the multiple cofactor $(\dots(F_{p1})_{p2}) \dots$ is denoted $F_{p1p2\dots}$. Cofactors are available in Aralia.

II.4 Quantifiers

Quantifiers are also available:

- Universal quantifier, denoted by $\forall v_1, \dots, v_n F$, where the v_i 's are variables and F is any formula. $\forall v_1, \dots, v_n F$ is equivalent $\forall v_1(\dots(\forall v_n F)\dots)$. $\forall v F$ is equivalent to $F[1/v].F[0/v]$.
- Existential quantifier, denoted by $\exists v_1, \dots, v_n F$, where the v_i 's are variables and F is any formula. $\exists v_1, \dots, v_n F$ is equivalent $\exists v_1(\dots(\exists v_n F)\dots)$. $\exists v F$ is equivalent to $F[1/v]+F[0/v]$.

Consider, for instance, the formula $F = ab + \bar{a}c$. $\forall a F$ and $\exists a F$ are defined as follows.

$$F[1/a] = 1.b + -1.c = b$$

$$F[0/a] = 0.b + -0.c = c$$

$$\forall a F = F[1/a].F[0/a] = b.c$$

$$\exists a F = F[1/a]+F[0/a] = b+c$$

II.5 Concrete Syntax for Boolean Formulae

The table 2.1 describes the concrete Aralia syntax for Boolean formulae. In this table, $v_1, v_2 \dots$ denote variables. $p_1, p_2 \dots$ denote literals, i.e. either variables v_i 's or their negations $-v_i$'s. Finally, $F_1, F_2 \dots$ denote formulae. Note that parentheses around formulae are mandatory.

Variables	$[a-zA-Z][a-zA-Z0-9-_.]^*$, i.e any sequence of letters, digits, -, _ or . characters beginning with a letter
Constants	0, 1
Not	$-F$
Or	$(F_1 \mid \dots \mid F_n)$
And	$(F_1 \& \dots \& F_n)$
Implications	$(F \Rightarrow G)$
Iff	$(F_1 = \dots = F_n)$
Xor	$(F_1 \# \dots \# F_n)$
If-Then-Else	$(F ? G : H)$
k -out-of- n	$@(k, [F_1, \dots, F_n])$
Cardinality	$\#(l, h, [F_1, \dots, F_n])$
Quantifiers	exists $v_1, \dots, v_n F$, forall $v_1, \dots, v_n F$
Cofactors	cofactor $v_1, \dots, v_n F$
Equations	$v := F;$

Table 2.1. Syntax of Aralia formulae

Example: Here follows three correct encodings of the formula $F = ab + \bar{a}c$ within the Aralia syntax.

$$F := ((a \& b) \mid (-a \& c)); \quad \left| \begin{array}{l} F := (F_1 \& F_2); \\ F_1 := (a \& b); \\ F_2 := (-a \& c); \end{array} \right| \quad \left| F := (a ? b : c); \right.$$

It is sometimes useful to build the disjunct or the conjunct of all the variables that have a given property. Aralia proposes a specific way to build associative commutative through the notion of selector. Table 2.2 presents this mechanism. This notion of selector is detailed in chapter X.

Or	[,<selector>]
And	[&,<selector>]
Iff	[=,<selector>]
Xor	[#,<selector>]
k-out-of-n	[@,k, <selector>]
cardinality	[#,l,h, <selector>]
Quantifiers	[exists, [<selector>], F], [forall, [<selector>], F]

Table 2.2. Aralia syntax for flat formulae

II.6 Terminology

Consider the following store:

```

r1 := (g1 & g2);
r2 := (g3 & g2);
g1 := (a | b);
g2 := (-a | c);
g3 := (b | d);

```

This store contains two roots (or output variables) ($r1$ and $r2$), five gates ($r1$, $r2$, $g1$, $g2$ and $g3$) and four leaves (or input variables) (a , b , c and d).
 $g1$ and $g2$ are the children of $r1$. $r1$ and $r2$ are the parents of $g2$. $g1$, $g2$, a , b , c are the descendants of $r1$. $g1$, $g2$, $r1$ and $r2$ are the ancestors of a .

II.7 Summary

Aralia deals with quantified Boolean formulae. It manages formulae as a set of equations. Each equation associates a formula to a variable that may be reused in other equations (up to the condition that there is no loop).

The notion of cofactor and Shannon decomposition play a central role in the Aralia mathematical framework.

III DATA STRUCTURES

Aralia provides four different data structures to encode Boolean functions:

- Stores of Boolean equations. Stores are presented chapter II. They are used to encode the textual descriptions of the models under study.
- Binary Decision Diagrams (BDD for short), a Binary Decision Diagram is a compact encoding of the truth table of a Boolean formula. The BDD associated with a formula is computed from the set of equations that describes this formula.
- Zero-Suppressed Binary Decision Diagrams (ZBDD for short). Zero-Suppressed BDD are BDD with a different semantics. They are used to encode minimal cutsets and prime implicants. A ZBDD is obtained either by applying a Minimal Cutsets algorithm from a BDD or by applying the MOCUS algorithm from Boolean equations.
- Sum-Of-Products (SoP for short). A SoP is an explicit representation of sets of minimal cutsets or prime implicants. ZBDDs are more compact than SoP but some operations, such as sorting the minimal cutsets, are impossible on this data-structure. SoP are obtained by expanding ZBDDs.

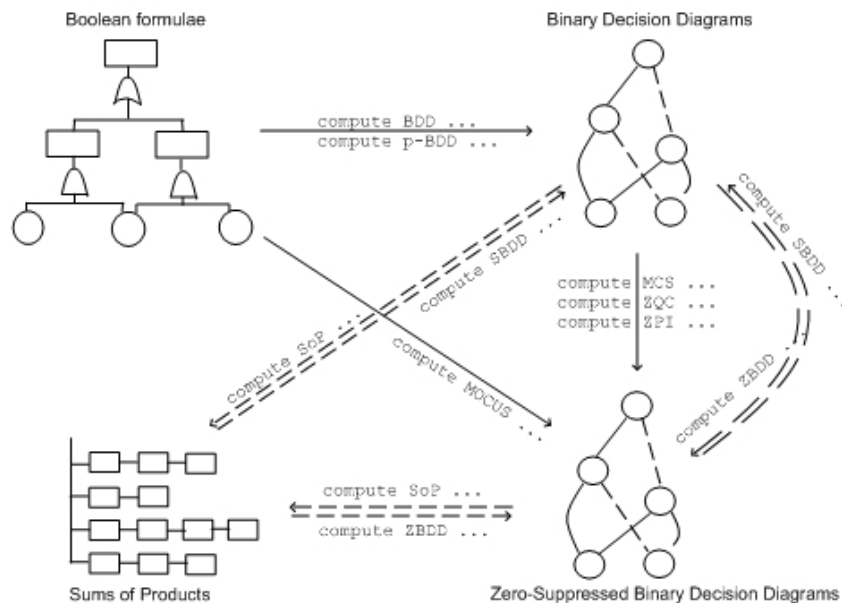


Figure 3.1. Aralia Data-Structures

Fig. 3.1 presents the four different data structures used by Aralia and the algorithms that compile one data structure into another.

Probabilistic measures (top event probability, importance factors...) can be assessed from either BDD, or ZBDD or SoP. However, algorithms designed for BDD provide exact answers while those designed for ZBDD and SoP provide only approximate results. Aralia makes transparent for the user the use of algorithms: the same commands can be applied on either data-structure.

III.1 Binary Decision Diagrams

Binary Decision Diagrams are a compact encoding of the truth tables of Boolean formulae. From the BDD that encodes a formula, it is possible to perform efficiently all of the probabilistic assessments (top event probability, importance factors...).

The BDD representation is based on the Shannon decomposition: Let F be a Boolean formula that depends on the variable v , then

$$F = v.F[v \leftarrow 1] + \bar{v}.F[v \leftarrow 0]$$

By choosing a total order over the variables and applying recursively the Shannon decomposition, the truth table of any formula can be graphically represented as a binary tree. The nodes are labelled with variables and have two out edges (a *then*-outedge, pointing to the node that encodes $F[1/v]$, and an *else*-outedge, pointing to the node that encodes $F[0/v]$). The leaves are labelled with either 0 or 1. The value of the formula for a given variable assignment is obtained by descending along the corresponding branch of the tree. The Shannon tree for the formula $F = ab + \bar{a}c$ and the lexicographic order is pictured Fig. 3.2 (dashed lines represent *else*-outedges).

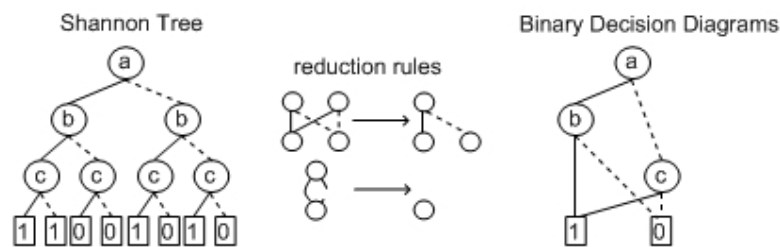


Figure 3.2. From the Shannon Tree to the BDD.

Indeed such a representation is very space consuming. It is however possible to shrink it by means of the following two reduction rules.

- Isomorphic subtrees merging. Since two isomorphic subtrees encode the same formula, at least one is useless.

- Useless nodes deletion. A node with two equal sons is useless since it is equivalent to its son ($F = v.F + \bar{v}.F$).

By applying these two rules as far as possible, one get the BDD associated with the formula. A BDD is therefore a directed acyclic graph. It is unique, up to an isomorphism. This process is illustrated on Fig. 3.2.

Logical operations (and, or, xor...) can be directly performed on BDDs. This results from the orthogonality of usual connectives and the Shannon decomposition:

$$(v.F1 + \bar{v}.F0) \oplus (v.G1 + \bar{v}.G0) = v.(F1 \oplus G1) + \bar{v}.(F0 \oplus G0)$$

where \oplus is any binary connective.

Among other consequences, this means that the complete binary tree is never built and then shrunk: the BDD encoding a formula is obtained by composing the BDDs encoding its subformulae. Moreover, a caching principle is used to store intermediate results of computations. This makes the usual logical operations (conjunction, disjunction) polynomial in the sizes of their operands.

Discussion: It is widely known, since the very first uses of BDDs, that the chosen variable ordering has a great impact on the size of BDDs, and therefore on the efficiency of the whole methodology. Finding the best ordering (or even a reasonably good one) is a very hard problem. Two kinds of heuristics are used to determine which variable ordering to apply. Static heuristics are based on topological considerations and select the variable ordering once for all. Dynamic heuristics change the variable ordering at some points of the computation. They are thus more versatile than the formers, but the price to pay is a serious increase of running times. Sifting is the most widely used dynamic heuristics.

The Aralia command to build the BDD associated with a formula is as follows.

```
compute BDD <variable-selector>;
```

It is worth noticing that this command computes the BDD of the selected variables and their descendants. Therefore, it suffices (and it is much better) to call it only on top events.

III.2 Zero-Suppressed Binary Decision Diagrams

Zero-Suppressed Binary Decision Diagrams are BDD with a different semantics for nodes (and slightly different reduction rules). They are used to encode for minimal cutsets and prime implicants.

A ZBDD encodes a set of products. Nodes are labelled with literals (and not just by variables). The semantics of ZBDD is as follows.

- The leaf 0 encodes the empty set: $\text{Set}[0] = \emptyset$.

- The leaf 1 encodes the set that contains only the empty product: $\text{Set}[1] = \{\{\}\}$.
- A node $\Delta(p, S_1, S_0)$, where p is a literal and S_1 and S_0 are two ZBDD encodes the following set of products.

$$\text{Set}[\Delta(p, S_1, S_0)] = \{\{p\} \cup \pi; \pi \in \text{Set}[S_1]\} \cup \text{Set}[S_0]$$

Commands to compute prime implicants and minimal cutsets are described chapter IV.

III.3 Sums of Products

Sum-Of-Products (SoP for short). A SoP is an explicit representation of sets of minimal cutsets or prime implicants. A SoP can be seen as a list of products, where each product is itself a list of literals. This representation is indeed much less compact than a ZBDD. However, it makes it possible to consider products individually. For instance, it is possible to sort literals inside products and products inside SoP. This operation is not allowed by the ZBDD representation.

Minimal cutsets (and prime implicants) algorithms in Aralia produce ZBDD as result. In order to expand a ZBDD into a SoP, the following command is to apply.

```
compute SoP from <source> [to <target>]
    <variable-selector> [<product-filter>];
```

In the above command, *<source>* is the name of the source data structure, *<target>* is the name of the target SoP. Product filters are discussed chapter XII.

III.4 Handles

BDD, ZBDD and SoP are accessed via so-called handles, themselves associated with variables. A variable (either a gate or a basic event) maintains a set of handles. Handles are named and thus can be selected. Commands to manage handles are as follows.

```
display handle <handle-selector> <variable-selector>
    [<redirection>];
clear handle <handle-selector> <variable-selector>;
```

Selectors for both handles and variables are described chapter X;

III.5 Bibliography

Binary Decision Diagrams (BDDs) were introduced by Bryant in two seminal articles:

- [Bry86] R. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677-691, August 1986.
- [BRB90] K. Brace, R. Rudell, and R. Bryant. Efficient Implementation of a BDD Package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*. IEEE 0738, 1990.

Reference [Bry86] introduces the data structure and basic operations. Reference [BRB90] describes the implementation of a BDD package. A large literature exists now on this topic, including some survey papers and textbooks to which the reader should refer to get more details about this technique, for instance:

- [Bry92] R. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24:293-318, September 1992.
- [Min96] S.-I. Minato. *Binary Decision Diagrams and Applications to VLSI CAD*. Kluwer Academics Publishers, 1996. ISBN 0-7923-9652-9.
- [MT98] C. Meinel and T. Theobald. *Algorithm and Data Structures in VLSI Design*. Springer Verlag, 1998. ISBN 3-540-64486-5.

IV MINIMAL CUTSETS AND PRIME IMPLICANTS

Minimal cutsets are the key stone of reliability studies. Intuitively, a minimal cutset is a minimal set of basic events that induces the realisation of the top event. This intuition is sufficient when applied to coherent fault trees. However, it is not correct when applied to non-coherent fault trees. For this later case, in order to capture the idea of minimal solution, the notion of prime implicant should be substituted for the notion of minimal cutset. Prime implicants are sets of literals i.e. they may contain negated variables. This does not mean however that the notion of minimal cutset is not mathematically founded. It is and a full understanding of this notion requires some algebraic developments.

Such a sound mathematical definition is given below. Aralia commands to compute minimal cutsets and prime implicants are described.

IV.1 Preliminary Definitions

Here follows some preliminary definitions.

A literal is either a variable v (positive literal), or its negation $\neg v$ (negative literal). v and $\neg v$ are said opposite. We write \bar{p} as the opposite of the literal p .

A product is a set of literals assimilated to the conjunction of its elements. Products are often written like words. For instance, the product $\{a, b, \neg c\}$ is written $ab\bar{c}$.

A minterm on a set of variables $V = \{v_1, \dots, v_n\}$ is a product which contains one and only one literal built on each variable of V . We write $minterms(V)$ for the set of minterms built on V . If V contains n variables, 2^n minterms can be built on V .

An assignment of a set of variables $V = \{v_1, \dots, v_n\}$ is a function σ from V to $\{0, 1\}$ that assigns a value (true or false) to each variable of V . Using the truth tables of connectives, assignments are extended into functions from formulae built over V to $\{0, 1\}$.

An assignment σ satisfies a formula F if $\sigma(F) = 1$. It falsifies F if $\sigma(F) = 0$.

There is a one-to-one correspondence between the assignments of V and the minterms built on V : a variable v occurs positively in the minterm π iff and only if $\sigma(v)=1$ in the corresponding assignment σ . E.g. the minterm $ab\bar{c}$ corresponds to the function σ such that $\sigma(a)=\sigma(b)=1$ and $\sigma(c)=0$, and vice-versa.

A formula F can always be interpreted as the set of minterms (built on the set $\text{var}(F)$ of its variables) that satisfy it.

For example, the formula $F = ab + \bar{a}c$ can be interpreted as the set $\{abc, ab\bar{c}, \bar{a}bc, \bar{a}\bar{b}c\}$.

For the sake of the convenience, we use set notations for formulae and minterms. E.g. we note $\sigma \in F$ when $\sigma(F) = 1$.

There exists a natural order over literals: $\neg v < v$. This order can be extended to minterms: $\pi \leq \rho$ iff for each variable v , $\pi(v) \leq \rho(v)$. A physical interpretation of \leq is that π contains less information than ρ for it realizes less basic events. E.g. $\bar{a}\bar{b}c \leq \bar{a}bc$ because a occurs negatively in $\bar{a}\bar{b}c$ and positively in $\bar{a}bc$.

From an algebraic viewpoint, the set $\text{minterms}(V)$ equipped with the above partial order forms a lattice, as illustrated Fig. 4.1. The order relation is represented by lines (bottom-up). For the sake of the simplicity, transitive relations are not pictured.

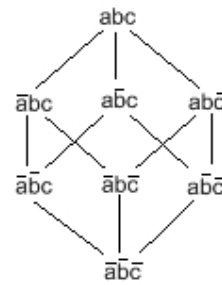


Figure 4.1. The lattice of minterms for $\{a, b, c\}$.

A formula F is said monotone if for any pair of minterms π and ρ such that $\pi \leq \rho$, then $\rho \in F$ implies that $\pi \in F$.

The formula $F = ab + \bar{a}c$ is not monotone because, $\bar{a}\bar{b}c \in F$, $\bar{a}\bar{b}c \leq \bar{a}bc$ but $\bar{a}bc \notin F$. This is graphically illustrated Fig. 4.2 (F minterms are black, the others are grey).

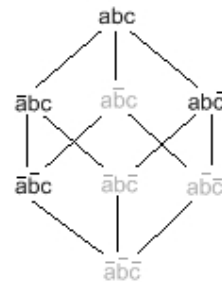


Figure 4.2. Minterms of $F = ab + \bar{a}c$.

Coherent fault trees, which are built over variables, and-gates, or-gates and k -out-of- n connectives, are monotone formulae. Non monotony is introduced by negations.

IV.2 Prime implicants and minimal cutsets

IV.2.1 Prime implicants

We can now introduce the notion of prime implicant.

A product π is an implicant of a formula F if for any minterm ρ such that $\pi \subseteq \rho$, $\rho \in F$.

An implicant π of F is prime if no proper subset of π is an implicant of F .

The set of prime implicants of F is denoted $PI[F]$.

For instance, the formula $F = ab + \bar{a}c$ admits the following set of prime implicants: $PI[F] = \{ab, \bar{a}c, bc\}$. Note that $\bar{a}b$ is an implicant of F because both $\bar{a}b\bar{c}$ and $\bar{a}bc$ satisfy F . It is prime because neither \bar{a} nor b are implicants of F .

IV.2.2 Minimal cutsets

In reliability models, there is, in general, a fundamental asymmetry between positive and negative literals. Positive literals represent unexpected (and often undesirable and rare) events such as failures. They are in some sense the only ones of interest. This is the reason why most of the fault tree assessment tools never produce minimal cutsets with negative literals. They produce only something related to positive parts of prime implicants.

To illustrate the above discussion, let us consider again the formula $F = ab + \bar{a}c$. We have $PI[F] = \{ab, \bar{a}c, bc\}$. This does correspond to the notion of minimal solution of F , but this does not correspond to the intuitive notion of minimal cutsets. The expected minimal cutsets are ab and c .

There are cases however where negative literals must be kept. Think for instance, to the case where a variable is used to model the night versus day opposition. So, we have to consider the set L of literals that convey interesting information. L is typically the set of all positive literals plus some negative literals.

From now, we shall say that a literal p is significant if it belongs to L and that it is critical if it is significant and while its opposite is not.

Let V be a finite set of variables, let L be a subset of literals that may built over V , finally let F be a formula built over V .

We shall define minimal cutsets of F as minimal solutions from which literals outside L are removed because they “do not matter”. Intuitively, a cutset is a product that

contains only literals from L and that can be completed with literals not in L in order to give an implicant of F .

A first way to define minimal cutsets is derived straight from the idea to keep only significant parts of prime implicants. Let $PI_L[F]$ be the set of products obtained first by removing from products of $PI[F]$ literals not in L and second by removing from the resulting set the non minimal products. Formally, $PI_L[F]$ is defined as follows.

$$PI_L[F] = \{\pi \cap L; \pi \in PI[F] \text{ and there is no } \rho \text{ in } PI[F] \text{ such that } \rho \cap L \subset \pi \cap L\}$$

This first definition captures the intuitive notion of minimal cutsets. For instance, it is easy to verify that $PI_{\{a,b,c\}}[ab+\bar{a}c] = \{ab, c\}$. This definition has the drawback to be based on the definition of prime implicants. This makes it not suitable to design an algorithm to compute minimal cutsets without computing prime implicants.

The second way to define minimal cutsets that avoids this drawback is as follows.

Let \leq_L be the binary relation among opposite literals defined as follows.

$$p \leq_L \bar{p} \text{ if } p \notin L$$

The comparator \leq_L is extended into a binary relation over $minterms(V)$ as follows.

$$\pi \leq_L \rho \text{ if for any variable } v, \pi[v] \leq \rho[v],$$

where $\pi[v]$ (resp. $\rho[v]$) denotes the literal built over v that belongs to π (resp. to ρ). Intuitively, $\pi \leq_L \rho$ when π is less significant than ρ .

The comparator \leq_L is both reflexive ($\pi \leq_L \pi$ for any π) and transitive ($\pi \leq_L \sigma$ and $\sigma \leq_L \rho$ implies $\pi \leq_L \rho$, for any π, σ and ρ). Therefore, \leq_L is a pre-order.

A product π over V is a cutset of F w.r.t. L if $p \subset \pi$ and for any minterm σ such that $\pi \leq_L \sigma$ there exists a minterm δ such that $\delta \leq_L \sigma$ and $\delta \in F$.

A cutset π is minimal if there is no cutset ρ such that $\rho \subset \pi$.

We denote by $MC_L[F]$ the set of minimal cutsets w.r.t. L of F .

Consider again the formula $F = ab + \bar{a}c$.

- If $L = \{a, \bar{a}, b, \bar{b}, c, \bar{c}\}$, minterms are pairwise incomparable. Therefore, $MC_L[F] = PI[F]$.
- If $L = \{a, b, c\}$, $ab\bar{c} \leq_L abc$, and $\bar{a}\bar{b}c \leq_L abc$, $\bar{a}\bar{b}c \leq_L abc$, $\bar{a}bc$, $\bar{a}bc$, therefore the cutsets of F w.r.t. L are abc , ab , ac , bc and c and $MC_L[F] = \{ab, c\}$.
- If $L = \{b, \bar{b}, c, \bar{c}\}$, the cutsets of F w.r.t. L are bc , b and c and $MC_L[F] = \{b, c\}$.

The two definitions of minimal cutsets are actually equivalent. Let F be a Boolean formula and let L be a set of literals built over $var(F)$. Then, the following equality holds.

$$PI_L[F] = MC_L[F]$$

Note finally that if $L=V$, a positive product π is a cutset if and only if the minterm $\pi \cup \{\bar{v}; v \in V \text{ and } v \notin \pi\}$ is an implicant of F .

IV.2.3 What do minimal cutsets characterize?

Any formula is equivalent to the disjunction of its prime implicants. A formula is not in general equivalent to the disjunction of its minimal cutsets. The widening operator ω_L gives some insights about the relationship between a formula F , its prime implicants and its minimal cutsets w.r.t. the subset L of the significant literals.

The operator ω_L is an endomorphism of the Boolean algebra $(minterms(V), \cap, \cup, \bar{})$ that associates to each set of minterms (formula) F the set of minterms ω_L defined as follows.

$$\omega_L = \{\pi; \exists \rho \text{ s.t. } \rho \leq_L \pi \text{ and } \rho \in F\}$$

Intuitively, ω_L enlarges F with all of the minterms that are more significant than a minterm already in F .

Consider again the formula $F = ab + \bar{a}c$.

- If $L = \{a, \bar{a}, b, \bar{b}, c, \bar{c}\}$, then, $\omega_L[F] = F$.
- If $L = \{a, b, c\}$, then $\omega_L[F] = abc + ab\bar{c} + \bar{a}bc + \bar{a}\bar{b}c$.
- If $L = \{b, \bar{b}, c, \bar{c}\}$, then $\omega_L[F] = abc + ab\bar{c} + \bar{a}\bar{b}c + \bar{a}b\bar{c}$.

The operator ω_L has a *number* of interesting properties that are summarized by the following facts.

ω_L is idempotent: $\omega_L(\omega_L(F)) = \omega_L(F)$.

$$PI[\omega_L(F)] = MC_L[F].$$

The above facts show that ω_L acts as a projection. Therefore, the formulae F such that $PI[F] = MC_L[F]$ are the fixpoints of ω_L , i.e. the formulae such that $\omega_L(F) = F$. If $L=V$, fixpoints are monotone formulae.

They give also a third way to define minimal cutsets: the minimal cutsets of a formula F are the prime implicants of a pessimistic approximation of F . This approximation is obtained by widening F with all of the minterms that are more significant, and therefore less expected, than a minterm already in F .

IV.2.4 Decomposition Theorems

The algorithms to compute prime implicants and minimal cutsets from BDDs rely on so-called decomposition theorems. These theorems use the Shannon decomposition as a basis.

The decomposition theorem for prime implicants is as follows.

Decomposition Theorem for Prime Implicants: Let $F = v \rightarrow F_1, F_0$ be a formula (such that F_1 and F_0 don't depend on v). Then, the set of prime implicants of F is the as follows.

$$PI[F] = PI_n \cup PI_1 \cup PI_0$$

where (" $/$ " stands for the set difference),

$$PI_n = PI[F_1.F_0]$$

$$PI_1 = \{v.\pi; \pi \in PI[F_1]/PI_n\}$$

$$PI_0 = \{\bar{v}.\pi; \pi \in PI[F_0]/PI_n\}$$

The decomposition theorem for minimal cutsets is as follows.

Decomposition Theorem for Minimal Cutsets: Let $F = v \rightarrow F_1, F_0$ be a formula (such that F_1 and F_0 don't depend on v). Let L be the set of relevant literals. Then, the set of minimal cutsets F is the as follows.

case 1: both v and its negation belong to L . In that case,

$$MCS[F] = MCS[F_1] \cup MCS[F_0]$$

case 2: v belongs to L , its negation does not. In that case, there are two ways to compute $MCS[F]$.

First decomposition:

$$MCS[F] = MCS_1 \cup MCS_0$$

where,

$$MCS_0 = MCS[F_0]$$

$$MCS_1 = \{v.\pi; \pi \in MCS[F_1+F_0]/MCS_0\}$$

Second decomposition:

$$MCS[F] = MCS_1 \cup MCS_0$$

where,

$$MCS_0 = MCS[F_0]$$

$$MCS_1 = \{v.\pi; \pi \in MCS[F_1] \div MCS_0\}$$

$$P \div Q = \{\pi \in P; \forall \rho \in Q, \rho \text{ is not included in } \pi\}$$

case 3: neither v nor its negation belong to L . In that case, the decomposition theorem is the same as for prime implicants.

IV.3 Commands to compute Prime Implicants and Minimal Cutsets

The command to compute prime implicants is as follows.

```
compute ZPI [! <order>] [to <handle>] <variable-selector>;
```

<variable-selector> indicates the variables for which the computation is to be performed.

! *<order>* is optional. It sets the maximum order of the computed prime implicants.

to *<handle>* is also optional. It sets the name of the ZBDD that encodes the prime implicants.

There are four available algorithms to compute minimal cutsets. The first one, *ZMC* works only for coherent models (monotone formulae). The set of relevant literals is assumed to be the set of all positive literals. The syntax *ZMC* is as follows.

```
compute ZMC [! <order>] [to <handle>] <variable-selector>;
```

The second (*ZPC*) and the third (*ZQC*) algorithm make the same assumption about relevant literals as *ZMC*. They are based respectively on the first and second decomposition (case 2 of MCS decomposition theorem). Their syntax is as follows.

```
compute ZPC [! <order>] [to <handle>] <variable-selector>;
```

```
compute ZQC [! <order>] [to <handle>] <variable-selector>;
```

In general, *ZQC* is more efficient than *ZPC*.

The next algorithm (*MCS*) is the most general one. It makes it possible to define the set of variables on which it uses the MCS decomposition theorem. Its syntax is as follows.

```
compute MCS <MCS-algorithm> [! <order>] [to <handle>]
    <variable-selector> <variable-selector>;
```


The first variable selector is to define the set of relevant literals. MCS-algorithm is an integer that defines the decomposition theorem to apply:

- 0 to use the same decomposition as ZPI.
- 1 to use the same decomposition as ZMC.
- 2 to use the same decomposition as ZPC.
- 3 to use the same decomposition as ZQC.

Finally, the last algorithm (ZWC) is based on the same decomposition theorem as ZQC. It is less efficient, but makes it possible to use filters to select minimal cutsets (see chapter XII for a presentation of filters).

Its syntax is as follows.

```
compute ZWC [from <handle>] [to <handle>]  
    <variable-selector> <filter> ;
```

If the number of selected cutsets is known to be small (say only several thousands), this algorithm is certainly the one to use.

IV.4 Summary

The commands to compute minimal cutsets are recalled table 4.3.

<i>Name</i>	<i>Syntax</i>
Prime implicants	compute ZPI [! <order>] [to <handle>] <variable-selector> ;
Minimal Cutsets	compute ZMC [! <order>] [to <handle>] <variable-selector> ;
Minimal Cutsets	compute ZPC [! <order>] [to <handle>] <variable-selector> ;
Minimal Cutsets	compute ZQC [! <order>] [to <handle>] <variable-selector> ;
Minimal Cutsets	compute MCS <MCS-algorithm> [! <order>] [to <handle>] <variable-selector> <variable-selector>;
Minimal Cutsets	compute ZWC [from <handle>] [to <handle>] <variable-selector> [<filter>]

Table 4.3. Commands to compute Prime Implicants and Minimal Cutsets

V HANDLING VERY LARGE MODELS

It could be the case that the model under study is so large that the Aralia engine is not able to deal with. Due to the highly exponential nature of the computations to be performed, there is sometimes nothing to do but to make the model simpler. Quite often however, a fine tuning of the Aralia engine makes it possible to tackle models that are beyond the scope of default parameters.

- The Aralia engine can be tuned in ways:
- By using variable ordering heuristics and by rewriting the model to make it easier to handle.
- By performing approximate computations, i.e. by computing p-BDD rather than BDD,
- By tuning sizes of BDD-tables.

This chapter is devoted to these various means to go a step further with the Aralia engine.

V.1 Tuning BDD Tables

The commands described in this section can be applied at any time during an Aralia session. However, they are better applied in the configuration file `'aralia.cfg'`.

V.1.1 *BDD indices and the BDD entry table*

In order to build the BDD of a formula, an index must be associated with each input variable (basic event). In the current version of Aralia, BDD indices are encoded onto a half machine word (16 bits). ZBDD require associating an index with each literal, i.e. two indices per variable. For technical reasons, indices of positive and negative literals must be respectively in the form $2n-1$ and $2n$ ($n \geq 1$). Since BDD and ZBDD share the same entry table, it is convenient to give the same value to the BDD index of a variable as to the ZBDD index of the positive literal of this variable. Therefore, the above n can range from 1 to $2^{14} = 16381$. Hence, the maximum number of basic events Aralia is able to deal with is 16381.

In practice, the user can not define BDD indices directly. Rather, he or she associates an entry number to each variable. This entry number must range from 1 to 16381. The BDD manager then defines automatically indices from entry numbers (dynamic reordering makes the correspondence non trivial). For unfortunate historical reasons, the entry number of a variable is accessed through the keyword `bdd-index`. The command to display entry numbers is as follows.

```
display bdd-index <variables> [<redirection>] ;
```

Note that the command `compute BDD` defines automatically entry numbers by means of a depth-first left most traversal of the formula, starting at 1. A similar indexing is obtained by means of the following command.

```
set bdd-indexing depth-left topEvent;
```

It is possible to defined entry numbers by means of the following command.

```
set bdd-indexing manual  
  <variable>@<entry-number> [, <variable>@<entry-number>]* ;
```

By default, the size of the BDD entry table is set to 8191. In order to handle more basic events (but still less than 16381), the following command should be put in the configuration file.

```
set bdd-entry-table size 16381 ;  
/* or any number less than 16381 */
```

The current size of the BDD entry table is obtained as follows.

```
display bdd-entry-table size [<redirection>] ;
```

V.1.2 The BDD Unique-table

BDD nodes are stored in a table so-called `bdd-unique-table`. This table is divided into pages. It is possible to set the size of the pages as well as the maximum number of pages. The size of the pages should be a power of two (for alignment reasons).

Commands to manage the BDD unique-table are the following.

```
set bdd-unique-table default-page-size <integer>;  
set bdd-unique-table maximum-page-number <integer>;  
display bdd-unique-table default-page-size [<redirection>];  
display bdd-unique-table maximum-page-number [<redirection>];
```

V.1.3 The BDD-Hashtables

In order to access BDD nodes, Aralia uses a hashtable per each literal (i.e. two per variables). This hashtable is created and resized automatically by Aralia. The minimum and maximum sizes are respectively 31 and 1023. Except for very special cases, it is not necessary to modify these values.

Commands to manage BDD-hashtables are as follows.

```
set bdd-hashtables minimum-size <integer>;
set bdd-hashtables maximum-size <integer>;
display bdd-hashtables minimum-size [<redirection>];
display bdd-hashtables maximum-size [<redirection>;
```

V.1.4 The BDD-Hashcache

The BDD technology relies on memorization of intermediate results. In this way, space is given to save time. Aralia manages a memorization table called `bdd-hashcache`. The larger this table, the faster the Aralia engine. The size of this table should be a prime number. It is automatically resized between a minimum size and a maximum size. To handle very large models it is a good idea to set the minimum and maximum sizes to the same large value. Each cell of the hashcache is made of 4 machine words, i.e. 16 bytes on 32 bits machines. Commands to manage the hashcache are as follows.

```
set bdd-hashcache minimum-size <integer>;
set bdd-hashcache maximum-size <integer>;
display bdd-hashcache minimum-size [<redirection>];
display bdd-hashcache maximum-size [<redirection>;
```

The following table gives good values for the hashcache.

2^n-1	size
10	1021
11	2039
12	4093
13	8191
14	16381
15	32749
16	65521
17	131071
18	262139
19	524287
20	1048573
21	2097143
22	4194301
23	8388593
24	16777213
25	33554393

VI STOCHASTIC LAYER

VI.1 Introduction

This chapter describes the stochastic layer of Aralia models. The stochastic layer is populated with failure probabilities or failure probability distributions associated with basic events. Probability distributions are described by (stochastic) expressions. These expressions may depend on parameters (variables), so the stochastic layer can be seen as a set of stochastic equations.

Stochastic expressions play actually two roles:

- They are used to associate a probability distribution with each basic event, i.e. for a given mission time t , the probability $Q(t)$ that the given basic event occurs before t . The probability distribution associated with a basic event is typically a negative exponential distribution of parameter λ :

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

Note that, the mission time t is a parameter of a special type.

- Parameters are sometimes not known with certainty. Sensitivity analyses are thus performed (by means of Monte-Carlo simulations) to study the change in risk due to this uncertainty. Expressions are therefore used to describe distributions of parameters. Typically, the parameter λ of a negative exponential distribution will be itself distributed according to a lognormal law of mean 0.001 and error factor 3, with a 95% confidence range.

Stochastic expressions are made of the following elements:

- Boolean and numerical constants,
- Stochastic variables, i.e. parameters, including the special variable to represent the mission time,
- Boolean and arithmetic operations (sums, differences, products...),
- Built-in expressions that can be seen as macro-expressions that are used to simplify and shorten the writing of probability distributions (E.g. exponential, Weibull...),
- Primitives to generate numbers at pseudo-random according to some probability distribution. The base primitive makes it possible to generate random deviates with a uniform probability distribution. Several other primitives are derived from this one to generate random deviates with normal, lognormal... distributions.

Moreover, it is possible to define discrete distributions “by hand” through the notion of histogram.

VI.2 Commands

Probability distributions are associated with basic events through the command “basic-event”.

```
basic-event set <selector> <parameter> ;
```

This command associates the given parameter (probability distribution) to the selected basic events.

```
basic-event clear <selector> ;
```

This command dissociates probability distributions from the selected basic events.

```
basic-event display <selector> [<redirection>] ;
```

This command displays the probability distributions associated with the selected basic events.

The following commands are used to manage named parameters.

```
parameter set <selector> <parameter> ;
```

This command defines the given parameters.

```
parameter clear <selector> ;
```

This command reset definitions of the given parameters.

```
parameter display definition <selector> [<redirection>] ;
```

This command displays the definitions of the given parameters.

```
parameter display value <selector> [<redirection>] ;
```

This command displays the current values of the given parameters.

```
parameter rename <identifier> <identifier>;
```

This command renames the parameter with the first name into the second name.

```
parameter reset <selector> ;
```

This command resets the given parameters to their mean values.

```
parameter draw <selector> ;
```

This command draws at pseudo-random the values of the given parameters (according to their definitions).

VI.3 Parameters

Probability Distributions associated with Basic Events can be defined as raw numbers, (arithmetic) combinations of parameters, time dependent distributions (with parameters) or random deviates (with parameters). Parameters can be named and reused at different places.

VI.3.1 *Constant Parameters*

Constant parameters are just floating point numbers, e.g.

```
basic-event set a 1.0e-3;
```

The symbol “pi” is evaluated as the constant “ $\pi=3.14159\dots$ ”.

VI.3.2 *Named parameters*

Named parameters are (stochastic) variables. They are defined by means of the command “parameter set”.

```
parameter set <identifier> <parameter>;
```

Example:

```
parameter set lambda 0.001;
parameter set mu mul + m2;
```

Parameters are managed into a store, in the same way Boolean equations are. It is therefore possible to select named parameters. A named parameter should be defined prior any use (although, by default, a named parameter takes the value 0).

VI.3.3 *Arithmetic operations on parameters*

Parameters can be defined as arithmetic operations on other parameters. The syntax for these parameters is as follows (parentheses are mandatory).

Operations	Syntax
minus	- <parameter>
addition	(<parameter> + ... + <parameter>)
subtraction	(<parameter> - ... - <parameter>)
multiplication	(<parameter> * ... * <parameter>)
division	(<parameter> / ... / <parameter>)
absolute value	abs(<parameter>)
trigonometric operations	op(<parameter>), where op is in cos, sin, tan, acos,

	asin, atan, cosh, sinh, tanh
exponential, power and logarithms	exp(<parameter>) pow(<parameter>, <parameter>) log(<parameter>) log10(<parameter>)
square root	sqrt(<parameter>)
closest integral values	ceil(<parameter>) floor(<parameter>)
modulus	mod(<parameter>, <parameter>)
minimum, maximum, mean	min([<parameter>, +]) max([<parameter>, +]) mean([<parameter>, +])

Example:

```
parameter set lambda (lambda1 + 3.0*lambda2 + lambda3);
parameter set mu (mu1 * mu2);
```

VI.3.4 Boolean Operations on Parameters

Boolean operations on parameters are useful mainly in relationship with conditional operations.

Operations	Syntax
not	not <parameter>
and	<parameter> and ... and <parameter>
or	<parameter> or ... or <parameter>
inequalities	<parameter> = <parameter> <parameter> # <parameter> <parameter> < <parameter> <parameter> > <parameter> <parameter> <= <parameter> <parameter> >= <parameter>

VI.3.5 Conditional Operations on Parameters

Two conditional operations on parameters are available: an if-then-else and a switch. Their syntax is as follows.

Operations	Syntax
if-then-else	ite(<parameter>, <parameter>, <parameter>)
switch	switch(case(<parameter>, <parameter>), ... case(<parameter>, <parameter>),

	<parameter>)
--	---------------

VI.4 Time-Dependent Distributions

VI.4.1 *Mission Time*

In general, the probability of occurrence of a basic event evolves through the time. Therefore, commands to compute probabilities and importance factors take one or several mission times as arguments. The mission time at which the calculation is performed intervenes into the definition of parameters through the special parameter “mission-time”.

VI.4.2 *Constant Distribution*

This probability distribution takes a single parameter: the probability q of the event.

$$Q(t) = q$$

Since $Q(t)$ does not depend on t , $w(t)=0$.

This distribution is useful to give a specific value to the probability of the event, or to study the solicitation of a component (e.g. valve opening, start of a diesel engine...).

VI.4.3 *Exponential Distribution*

This distribution takes two parameters: the failure rate λ of the component (that is assumed to be constant throughout the time) and the mission time t . Its definition is as follows.

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

In this case, $w(t) = \lambda \times e^{-\lambda t}$.

This distribution is widely used because it is almost the only one that makes it possible to get analytical results. Moreover, it is a good model of component life, at least when there are a large number of components. Since the failure rate does not vary, this distribution is well suited to model the life of many components after their infantile death period.

The syntax for this distribution is as follows.

```
exponential( <parameter>, <parameter> )
```

VI.4.4 GLM Distribution

GLM stands for Gamma-Lambda-Mu. This distribution generalizes the exponential distribution. It makes it possible to take into account repairable components (through the repairing rate μ) and failures on demand (through the probability γ of such an event). It takes four parameters, γ , the failure rate λ , μ and the mission time t (in this order). Its definition is as follows.

$$Q(t) = \frac{\lambda}{\lambda + \mu} - \frac{\lambda - \delta(\lambda + \mu)}{\lambda + \mu} \times e^{-(\lambda + \mu)t}$$

In this case, $w(t) = (1 - Q(t)) \times \lambda$.

The syntax for this distribution is as follows.

```
GLM( [<parameter> , ] 4 )
exponential( [<parameter> , ] 4 )
```

VI.4.5 MTT Distribution

MTT distribution is another form of the exponential distribution with two parameters: the $MTTF = 1/\lambda$, the $MTTR = 1/\mu$ and the mission time t (in this order). Its definition is as follows.

$$Q(t) = \frac{\lambda}{\lambda + \mu} \left[1 - e^{-(\lambda + \mu)t} \right]$$

In this case, $w(t) = (1 - Q(t)) \times 1/MTTF$.

The syntax for this distribution is as follows.

```
MTT( [<parameter> , ] 3 )
```

VI.4.6 Weibull Distribution

This distribution takes three parameters: a scale parameter α , a shape parameter β and the mission time t (in this order). Its definition is as follows.

$$Q(t) = 1 - \exp \left[- \left(\frac{t}{\alpha} \right)^\beta \right]$$

Even if $w(t)$ could be estimated, Aralia considers that it equals 0.

The Weibull distribution is very interesting because many experimental distributions can be represented, just by tuning parameters. For instance, if $\beta < 1$ the failure rate is

decreasing and the distribution is well suited to model the infant death or the debugging period of a component. On the other hand, if $\beta > 1$, the failure rate is increasing and the distribution is well suited to model components at the end of their life. Finally, if $\beta = 1$ the Weibull distribution is equivalent to an exponential distribution.

The syntax for this distribution is as follows.

```
Weibull( [ <parameter> , ] 3 )
```

VI.4.7 Dirac Distribution

The Dirac distribution takes one parameter: a delay d . The event occurs at d . Therefore, $Q(t) = 0$ if $t < d$ and $Q(t) = 1$ if $t \geq d$. $w(t)$ is set to 0.

The syntax of this distribution is as follows.

```
Dirac( <parameter> )
```

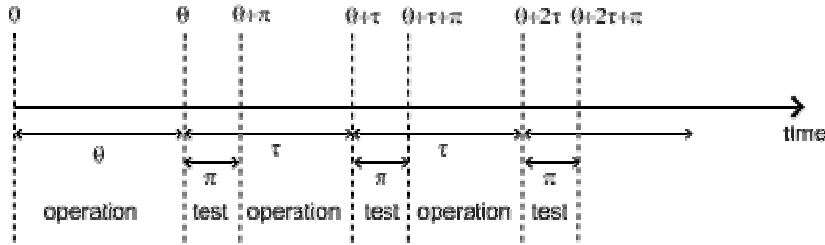
VI.4.8 Distributions for Periodically Tested Components

The “periodic-test” distribution is designed to get an as precise as possible assessment of the unavailability of a periodically tested component. This distribution takes the following parameters (in order).

1	λ	Failure rate when the component is working.
2	λ^*	Failure rate when the component is tested.
3	μ	Repair rate (once the test showed that the component is failed).
4	τ	Delay between two consecutive tests.
5	θ	Delay before the first test.
6	γ	Probability of failure due to the (beginning of the) test.
7	π	Duration of the test.
8	x	Indicator of the component availability during the test (1 available, 0 unavailable).
9	σ	Test covering: probability that the test detects the failure, if any.
10	ω	Probability that the component is badly restarted after a test or a repair.
11	t	mission time

In this case, Aralia sets w to λ .

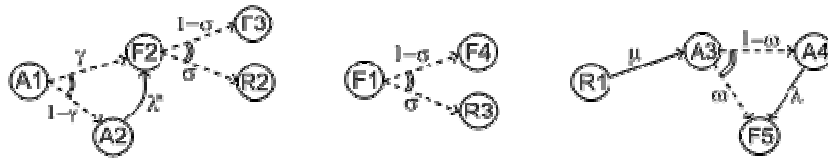
The following scheme illustrates the meaning of the parameters τ , θ and π .



There are three phases in the behaviour of the component. The first phase corresponds to the time from 0 to the date of the first test, i.e. θ . The second phase is the test phase. It spreads from times $\theta+n.\tau$ to $\theta+n.\tau+\pi$, with n any positive integer. The third phase is the functioning phase. It spreads from times $\theta+n.\tau+\pi$ from $\theta+(n+1).\tau$.

In the first phase, the distribution is a simple exponential distribution of parameter λ .

The component may enter in the second phase in three states, either working, failed or in repair. In the latter case, the test is not performed. The Markov graphs for each of these cases are pictured below.



A_i 's, F_i 's, R_i 's states correspond respectively to states where the component is available, failed and in repair. Dashed lines correspond to immediate transitions. Initial states are respectively A_1 , F_1 and R_1 .

The situation is simpler in the third phase. If the component enters available this phase, the distribution follows an exponential distribution of parameter λ . If the component enters failed in this phase, it remains phase up to the next test. Finally, the Markov graph for the case where the component is in repair is the same as in the second phase.

The syntax for this distribution is as follows.

```
periodic-test( [<parameter> , ] 11)
```

Aralia provides also two simplified forms for the periodic test distribution.

The first one takes five parameters: λ , μ , τ , θ and the mission time t . The test is assumed to be instantaneous. Therefore, parameters λ^* (the failure rate during the test) and x (indicator of the component availability during the test) are meaningless. There other parameters are set as follows.

- γ (the probability of failure due to the beginning of the test) is set to 0.
- σ (the probability that the test detects the failure, if any) is set to 1.
- ω (the probability that the component is badly restarted after a test or a repair) is set to 0.

The second one takes only four parameters: λ , τ , θ and the mission time t . The repair is assumed to instantaneous (or equivalently the repair rate $\mu = +\infty$).

The syntax for these simplified periodic test distributions are as follows.

```
periodic-test (
    <parameter>, /*  $\lambda$  */
    <parameter>, /*  $\mu$  */
    <parameter>, /*  $\tau$  */
    <parameter>, /*  $\theta$  */
    <parameter>) /*  $t$  */)
```

```
periodic-test (
    <parameter>, /*  $\lambda$  */
    <parameter>, /*  $\tau$  */
    <parameter>, /*  $\theta$  */
    <parameter>) /*  $t$  */)
```

VI.4.9 Dormant Distribution

The dormant distribution takes three parameters: a failure rate λ , the mean time to repair $MTTR$ and a delay d (in this order). Its value does not depend on the mission time t . It is as follows.

$$Q(t) = \frac{\lambda d - (1 - e^{-\lambda d}) + \lambda \cdot MTTR \cdot (1 - e^{-\lambda d})}{\lambda d + \lambda \cdot MTTR \cdot (1 - e^{-\lambda d})}$$

Since $Q(t)$ does not depend on t , $w=0$.

The syntax for this distribution is as follows.

```
dormant( [ <parameter> , ] 3 )
```

VI.4.10 Standby Distribution

A Standby Model may be used to represent the failure and repair characteristics of a redundant subsystem. Blocks associated with a Standby Model represent a group of components, including some that may be actively running and some that may lie dormant. A Standby Model requires the following parameters to be specified.

- The operating failure rate, i.e. the failure rate of the components when they are actively in use

- Standby failure rate, i.e. the failure rate of the components when they are in Standby mode.
- The repair rate of the components.

The failure and repair rates of the components are assumed to be constant. Thus it is important that the total number of components in the subsystem be specified together with the total number of normally active operating components. If fewer components than the specified number of operating components are available, then the standby subsystem is considered to be unavailable. The number of repair crews available indicates the maximum number of components in the subsystem that may be repaired at the same time. Note that the calculated unavailability for the Standby Model is the steady-state value.

Standby redundancies must satisfy the following requirements:

- The standby redundancy consists of n identical components.
- The redundant configuration has m ($\leq n$) principal components.
- At most, r (≥ 1) components can be repaired at a time.

The following differential equations apply:

$$\dot{P}_{(0)} = -\lambda_0 P_{(0)} + \mu_1 P_{(1)}$$

$$\dot{P}_{(k)} = \lambda_{k-1} P_{(k-1)} - (\lambda_k + \mu_k) P_{(k)} + \mu_{k+1} P_{(k+1)}$$

$$\dot{P}_{(n)} = \lambda_{n-1} P_{(n-1)} - \mu_n P_{(n)}$$

where, k is the number of components under repair.

$$\lambda_k \equiv m\lambda + (n - m - k)\bar{\lambda}, \quad \text{for } k = 0, \dots, n - m$$

$$\lambda_k \equiv (n - k)\lambda, \quad \text{for } k = n - m + 1, \dots, n - 1$$

$$\mu_k \equiv \min\{r, k\} \times \mu, \quad \text{for } k = 1, \dots, n$$

The parameter $Q(t)$ is given by $Q(t) = P_{(n-m+1)} + \dots + P_{(n)}$

$$P_k = \frac{\lambda_{k-1}}{\mu_k} P_{(k-1)} = \frac{\lambda_0 \lambda_1 \dots \lambda_{k-1}}{\mu_1 \mu_2 \dots \mu_k} P_{(0)} \equiv \theta_k P_{(0)}, k = 1, \dots, n$$

Because the sum of all probabilities is equal to unity, the following equality holds.

$$P_{(k)} = \theta_k / (\theta_0 + \theta_1 + \dots + \theta_n)$$

The syntax for this distribution is as follows.

standby([<parameter> ,] 6)

with the following meanings for parameters.

1	n	Number of components
2	m	Number of principal (active) components
3	r	Maximum number of components that can be repair at the same time
4	λ	Failure rate of operational components
5	$\bar{\lambda}$	Failure rate of standby components
6	μ	Repair rate of components

VI.4.11 Bound Time Distribution

The bound time distribution is a distribution modifier rather than a distribution. It takes three parameters: a start time t_0 , a period d and a probability distribution L (in this order). Its definition is as follows.

$$Q(t) = \begin{cases} 0 & \text{if } t \leq t_0 \\ Q_L(t - t_0) & \text{if } t_0 \leq t \leq t_0 + d \\ Q_L(d) & \text{if } t \geq t_0 + d \end{cases}$$

where $Q_L(t)$ denotes the unavailability of the component at time t assessed according the distribution L . The value of unconditional failure intensity is as follows.

$$w = \begin{cases} 0 & \text{if } t \leq t_0 \\ w_L & \text{if } t_0 \leq t \leq t_0 + d \\ 0 & \text{if } t \geq t_0 + d \end{cases}$$

This distribution (modifier) is useful to model components that are not started at the beginning of the mission but later on.

The syntax for this distribution is as follows.

```
bound-time( <parameter>, <parameter>, <distribution>)
```

VI.5 Random Deviates

VI.5.1 Uniform Deviates

Uniform parameters vary according to a uniform distribution into a given range defined by its lower- and upper-bounds. The default value of a uniform parameter is the mean of the range, i.e. $(\text{lower-bound} + \text{upper-bound})/2$.

The syntax for these parameters is as follows.

```
uniform-deviate(<parameter>, <parameter>)
```


Example:

```
basic-event set a exponential(lambda,mission-time);
parameter set lambda uniform(0.001,0.002);
```

VI.5.2 Normal Deviates

Normal deviates vary according to a normal distribution defined by its mean and its standard deviation (in this order). By default, the value of a normal deviate is its mean.

The syntax for these parameters is as follows.

```
normal(<parameter>, <parameter>)
```

Example:

```
basic-event set exponential(lambda,mission-time);
parameter set lambda normal(0.001,0.01);
```

VI.5.3 Lognormal Deviates

Lognormal parameters vary according to a lognormal distribution defined by its mean and error factor. The confidence level is given as third parameter.

The syntax for these parameters is as follows.

```
lognormal(<parameter>, <parameter>, <parameter>)
```

Example:

```
basic-event set exponential(lambda,mission-time);
parameter set lambda lognormal(0.001,3,0.95);
```

A random variable is distributed according to a lognormal deviate if its logarithm is distributed according to a normal deviate. If μ and σ are respectively the mean and the standard deviation of the relevant normal law, the probability density of the random variable is as follows.

$$f(x) = \frac{1}{\sigma \cdot x \cdot \sqrt{2\pi}} \times \exp \left[-\frac{1}{2} \left(\frac{\log x - \mu}{\sigma} \right)^2 \right]$$

Its mean $E(x)$ is as follows.

$$E(x) = \exp \left[\mu + \frac{\sigma^2}{2} \right]$$

The confidence intervals $[X_{0,05}, X_{0,95}]$ associated with a confidence level of 0.95 and the median $X_{0,50}$ are the following:

$$\begin{aligned} X_{0,05} &= \exp[\mu - 1.645\sigma] \\ X_{0,95} &= \exp[\mu + 1.645\sigma] \\ X_{0,50} &= \sqrt{X_{0,05} \times X_{0,95}} = e^{\mu} \end{aligned}$$

The error factor EF is defined as follows.

$$EF = \sqrt{\frac{X_{0,95}}{X_{0,05}}} = e^{1.645\sigma}$$

$$\text{with } \sigma = \frac{\log FE}{1.645} \text{ and } \mu = \log E(x) - \frac{\sigma^2}{2}.$$

Once the mean and the error factor are known, it is then possible to determine the confidence interval and thereby the parameters of the lognormal distribution.

VI.5.4 Histograms

Finally, parameters can be described as histograms, i.e. as lists of pairs (x_i, y_i) such that $0 < x_i < x_j$ for any $i < j$. Let $H: (x_1, y_1), \dots, (x_n, y_n)$ be such a histogram. The probability that H takes the value y_i is as follows.

$$p(H=y_i) = \frac{(x_i - x_{i-1})}{x_n}$$

where $x_0 = 0$. The mean value of H is as follows.

$$\text{mean}(H) = \frac{1}{x_n} \times \sum (x_i - x_{i-1}) \cdot y_i$$

The syntax for histograms is as follows.

histogram x1:y1 , ..., xn:yn

Example:

```
basic-event set a exponential(lambda,mission-time);
parameter set lambda histogram( 1:0.001, 2:0.002, 3:0.004);
```

VI.6 Summary

The available probability laws are recalled Table 6.1. The available probability law parameters are given Table 6.2.

<i>Name</i>	<i>Syntax</i>
Exponential	exponential([<parameter> ,]2)
Gamma-Lambda-Mu	exponential([<parameter> ,]4)
Weibull	GLM([<parameter> ,]4)
Periodic-test	Weibull([<parameter> ,]3)
	periodic-test([<parameter> ,]4)
	periodic-test([<parameter> ,]5)
	periodic-test([<parameter> ,]11)
Dormant	dormant([<parameter> ,]4)
Standby	standby([<parameter> ,] 6)
Bound-time	bound-time([<parameter> ,]3)

Table 6.1. Available time-dependent probability distributions

<i>Name</i>	<i>Syntax</i>
Uniform	uniform-deviate(<parameter>,<parameter>)
Normal	normal-deviate(<parameter>,<parameter>)
Lognormal	lognormal-deviate([<parameter> ,]3)
Histogram	histogram(<float> [, bin(<float>:<float>)]+)

Table 6.2. Available random deviates

parameter set <selector> <parameter> ;
parameter clear <selector> ;
parameter display definition <selector> [<redirection>] ;
parameter display value < selector> [<redirection>] ;
parameter reset <selector> ;
parameter draw <selector> ;

Table 6.3. Commands to manage parameters

VII PROBABILITY AND IMPORTANCE FACTORS

Aralia computes probabilities of (top) gates and importance factors of components from the data structures that encode models, i.e. BDD, ZBDD and SoP. Since these data structures are based on different principle, algorithms used in each case are different. Exact computations are performed on BDD (based on the Shannon Decomposition). Rare events approximation is applied on ZBDD and SoP.

All the classical importance factors $IF(S,e)$, where S is a system and e is a component, can be defined in terms of the probability $p(S)$ that the system fails, $p(e)$ that component fails and the conditional probability $p(S|e)$ that the system fails given that the component failed. Since different algorithms are used to assess probabilities, definitions of importance vary according to the data structure.

VII.1 Probabilities of Gates

The algorithm to compute the probability of a gate from a BDD is based on the Shannon Decomposition. I.e.

$$\begin{aligned} \text{BDD-Pr}(0) &= 0.0 \\ \text{BDD-Pr}(1) &= 1.0 \\ \text{BDD-Pr}(v.F_1 + \bar{v}.F_0) &= p(v).\text{BDD-Pr}(F_1) + (1-p(v)).\text{BDD-Pr}(F_0) \end{aligned}$$

As a consequence, the result is exact.

The algorithm to compute the probability of a gate from a ZBDD is based on the rare events approximation. I.e.

$$\text{ZBDD-Pr}(F) = \sum_{\pi \in F} p(\pi)$$

where $p(\pi)$ is just the product of the probability of basic events occurring in the cutset π .

The algorithm to compute the probability of a gate from a SoP is basically the same as the algorithm used for ZBDD. However, it is possible to compute subsequent terms of the Sylvester-Pointcaré development.

$$\text{SOP-Pr}(F) = \sum_{\pi \in F} p(\pi) - \sum_{\pi_1, \pi_2 \in F} p(\pi_1.\pi_2) + \dots + 1^{-(\text{order} \bmod 2 + 1)} \times \sum_{\pi_1, \dots, \pi_{\text{order}} \in F} p(\pi_1 \dots \pi_{\text{order}})$$

VII.2 Positive Probabilities of Gates

The algorithm to compute the positive probability of a gate from a BDD is based on the Shannon Decomposition, ignoring negative parts. I.e.

$$\begin{aligned} \text{BDD-PP}(0) &= 0.0 \\ \text{BDD-PP}(1) &= 1.0 \\ \text{BDD-PP}(v.F_1 + \bar{v}.F_0) &= p(v).\text{BDD-PP}(F_1) + \text{BDD-PP}(F_0) \end{aligned}$$

The algorithm to compute the positive probability of a gate from a ZBDD is based on the rare events approximation. I.e.

$$\text{ZBDD-PP}(F) = \sum_{\pi \in F} PP(\pi)$$

where $PP(\pi)$ is just the product of the probability of basic events occurring positively in the cutset π .

The algorithm to compute the probability of a gate from a SoP is basically the same as the algorithm used for ZBDD. However, it is possible to compute subsequent terms of the Sylvester-Pointcaré development.

$$\begin{aligned} \text{SOP-PP}(F) = \\ \sum_{\pi \in F} PP(\pi) - \sum_{\pi_1, \pi_2 \in F} PP(\pi_1.\pi_2) + \dots + 1^{-(\text{order} \bmod 2 + 1)} \times \sum_{\pi_1 \dots \pi_{\text{order}}} PP(\pi_1 \dots \pi_{\text{order}}) \end{aligned}$$

VII.3 Conditional Probabilities

The computation of importance factors rely on the computation conditional probabilities. Conditional probabilities $p(S|e)$ and $p(S|\bar{e})$, where S is a gate and e is a basic event, are available in Aralia. They are computed in different ways depending on the data structure from which they are assessed.

BDD makes it possible to compute exactly the conditional probability (thanks again to the Shannon decomposition).

$$\begin{aligned} \text{BDD-Pr}(0|e) &= 0.0 \\ \text{BDD-Pr}(1|e) &= 1.0 \\ \text{BDD-Pr}(v.F_1 + \bar{v}.F_0|e) &= p(v).\text{BDD-Pr}(F_1|e) + (1-p(v)).\text{BDD-Pr}(F_0|e) \\ \text{BDD-Pr}(v.F_1 + \bar{v}.F_0|v) &= \text{BDD-Pr}(F_1|v) \\ \text{BDD-Pr}(v.F_1 + \bar{v}.F_0|\bar{v}) &= \text{BDD-Pr}(F_0|\bar{v}) \end{aligned}$$

The algorithm to compute a conditional probability from a ZBDD is based on the rare events approximation. I.e.

$$\text{ZBDD-Pr}(S|e) = \sum_{e, \pi \in S} p(\pi) + \sum_{\pi \in S, e \notin \pi, \bar{e} \notin \pi} p(\pi)$$

$$\text{ZBDD-Pr}(S|\bar{e}) = \sum_{\bar{e}, \pi \in S} p(\pi) + \sum_{\pi \in S, e \notin \pi, \bar{e} \notin \pi} p(\pi)$$

The algorithm to compute the conditional probability from a SoP is basically the same as the algorithm used for ZBDD. However, it is possible, here again, to compute subsequent terms of the Sylvester-Pointcaré development.

VII.4 Marginal Importance Factor

The marginal importance factor, denoted by $MIF(S, e)$, is defined as follows.

$$MIF(S, e) = \frac{\partial(p(S))}{\partial(p(e))}$$

MIF is often called Birnbaum importance factor in the literature. It can be interpreted, when S is a monotone function, as the conditional probability that, given that e occurred, the system S is failed and e is critical, i.e. a repair of e makes the system working.

The following equalities hold.

$$MIF(S, e) = \frac{\partial(p(S))}{\partial(p(e))} \quad (1)$$

$$= p(S[1/e].\overline{S[0/e]}) \quad (2)$$

$$= p(S[1/e]) - p(S[0/e]) \quad (3)$$

Equality (2) holds only in the case of monotone functions (see below). Equality (3) holds because $p(S) = p(e).(p(S[1/e]) - p(S[0/e]) + p(S[0/e]))$. This equality is used to compute $MIF(S, e)$ in the case where S is encoded by a ZBDD or a SoP.

In the case S is encoded by a BDD, a numerical derivation algorithm is used.

Let V be the set of variables of S and let us denote $\text{crit}(S, e)$ the set of critical states of S w.r.t. e . $\text{crit}(S, e)$ is defined as follows.

$$\text{crit}(S, e) = \{e, \pi \in \text{minterms}(V); e, \pi \in S \text{ and } \bar{e}, \pi \notin S\}$$

$\text{crit}(S, e)$ is a Boolean function. Another way to write it is as follows.

$$\text{crit}(S, e) = e.(S[1/e].\overline{S[0/e]})$$

Therefore, $MIF(S,e)$ can be interpreted as the probability to be in a critical state w.r.t. e (divided by the probability of e). This latter probability is even a better candidate for the definition of $MIF(S,e)$ than the partial derivative: it can be extended in a smooth way to non-basic events, while the partial derivative cannot.

VII.5 Critical Importance Factor

The criticality of a component is related to the potential improvement of the system reliability resulting from the improvement of the component reliability. It is clear that it would be more difficult and costly to improve the more reliable components than to improve the less reliable ones. However, the marginal importance factor does not depend on the component reliability. The critical importance factor, denoted by $CIF(S,e)$, is another measure of component criticality that does depend on component reliability. It is defined as follows.

$$CIF(S,e) = \frac{p(e)}{p(S)} \times MIF(S,e)$$

In the case of monotone systems, $CIF(S,e)$ can be interpreted as the conditional probability that the system is in a critical state w.r.t. e , given that the system is failed.

VII.6 Diagnostic Importance Factor

The diagnostic importance factor, denoted by $DIF(S,e)$, is defined as follows.

$$DIF(S,e) = p(e|S) = \frac{p(e) \times p(S|e)}{p(S)}$$

$DIF(S,e)$ is often called Fussel-Vesely Importance factor. $DIF(S,e)$ is the fraction of the system unavailability (or risk) that involves the component failure. It is worth noticing that this interpretation still works in the cases where S is not monotone and/or e is not a terminal event.

VII.7 Risk Achievement Worth

The risk achievement worth, denoted by $RAW(S,e)$, is defined as follows.

$$RAW(S,e) = \frac{p(S|e)}{p(S)}$$

$RAW(S,e)$ is also called risk increase factor. It measures the increase in system failure probability assuming the worst case of failing component. It is an indicator of the importance of maintaining the current level of reliability for the component.

VII.8 Risk Reduction Worth

The risk reduction worth, denoted by $RRW(S,e)$, is defined as follows.

$$RRW(S,e) = \frac{p(S)}{p(S|\bar{e})}$$

$RRW(S,e)$ is also called risk decrease factor. It represents the maximum decreasing of the risk it may be expected by increasing the reliability of the component. Therefore this quantity may be used to select components that are the best candidates for efforts leading to improving system reliability.

VII.9 Aralia Commands

Aralia makes it possible to compute the probability of any gate or basic event. The command to do so is as follows.

```
compute <Pr-selector> [from <handle-selector>]
    <variable-selector>
    [<time-schedule>] [<tries>] [<doreset>] [<order>]
    [<redirection>] ;
```

The parameters of the above command are as follows.

- *<handle-selector>* is a selector of data structure handles (i.e. BDD, ZBDD or SoP names) on which the computation is to be performed. By default, the computation is performed on the BDD.
- *<variable-selector>* is a selector of variables for which the computation is to be performed.
- *<time-schedule>* sets the mission times at which the computation is to be performed. There are two ways to set mission times:
 - at *<time1>*, *<time2>*, ... i.e. a list of floating point numbers separated with commas.
 - from *<first-time>* to *<last-time>* step *<time-increment>*
- *<tries>* sets the number of tries (for Monte-Carlo simulations and sensitivity analyses, see chapter IX). The syntax for this option is as follows.
 - tries *<number-of-tries>*
- *<doreset>* if this option is set to 1, parameters of probability laws are no reset to their mean value. The syntax for this option is follows.
 - reset {0,1}

- `<order>` sets the order of the Sylvester-Pointcaré development (for SoP only).
- `<redirection>` is a redirection directive to print results into a text file. The syntax for redirection is as follows.
 - `> "file-name" /* overwrite the file */`
 - `>> "file-name" /* append results to the file */`

Examples:

```
compute Pr top_event;
compute Pr top_event at 10, 100, 1000;
compute Pr top_event from 10 to 100 step 10;
compute Pr from SoP,ZQC top_event;
compute Pr from SoP top_event,other_event > "results";
```

The command to compute importance factors is very similar to the command probabilities.

```
compute <IF-selector> [from <handle-selector>]
  <system-selector> <component-selector>
  [<time-schedule>] [<tries>] [<doreset>] [<order>]
  [<redirection>] ;
```

`<system-selector>` and `<component-selector>` are variable selectors. `<IF-selector>` is a selector for probabilistic quantities. Keywords for importance factors are the following.

- Pr: probability (in this case components are ignored).
- PP: probability taking into account only positive literals (in this case components are ignored).
- CPr: conditional probability $p(S|e=1)$.
- CQr: conditional probability $p(S|e=0)$.
- MIF: marginal importance factor.
- CIF: critical importance factor.
- DIF: diagnostic importance factor.
- RAW: risk achievement worth.
- RRW: risk reduction worth.

Examples:

```
compute Pr,MIF top_event e1,e2;
compute Pr top_event e1, e2 at 10, 100, 1000;
compute MIF,CIF,DIF,RAW,RRW top_event e1,e2
  from 10 to 100 step 10;
compute MIF,CIF,RAW from SoP,ZQC top_event leaves(top_event);
compute RAW,RRW from SoP top_event,other_event e1 > "results";
```

VII.10 Summary

The commands to compute importance factors are recalled Table 7.1.

<i>Name</i>	<i>Syntax</i>
Probability : Pr PP	compute <Pr-selector> [from <handle-selector> <system-selector> [<time-schedule>] [<tries>] [<doreset>] [<order>] [<redirection>] ;
Importance Factors : CPr CQr MIF CIF DIF RAW RRW	compute <IF-selector> [from <handle-selector> <system-selector> <component-selector> [<time-schedule>] [<tries>] [<doreset>] [<order>] [<redirection>] ;

Table 7.1. Commands to compute importance factors

VIII TIME DEPENDENT ANALYSES

The Fault Tree method (and more generally all methods based on Boolean representations) makes it possible to assess the availability at time t of the system under study. However, it is much more interesting in practice to get system reliability or failure rate at t . The study of system reliability and failure rate is often called “time dependent analyses”.

No universal method exists to assess these two parameters (except to transform the model into a Markov graph, or to perform Monte-Carlo simulation). Aralia implements several algorithms that give approximate results. This chapter presents these algorithms.

VIII.1 Mathematical Definitions of System Reliability

Let S denote the system under study. Let T denote the date of the first failure of S . T is a random variable. It is called the lifetime of S . We assume that components of S where as good as new at time 0 and that they are as good as new after a repair.

Reliability $R_S(t)$ and unreliability $F_S(t)$: the reliability of S at t is the probability that S experiences no failure during time interval $[0, t]$, given that it all its components were working at 0. Formally,

$$R_S(t) = \Pr\{t < T\} \quad (1)$$

The unreliability is just the opposite.

$$F_S(t) = \Pr\{t \geq T\} = 1 - R_S(t) \quad (2)$$

The curve $R_S(t)$ is a survival distribution. This distribution is monotonically decreasing. Moreover, the following asymptotic properties hold.

$$\lim_{t \rightarrow 0} R_S(t) = 1 \quad (3)$$

$$\lim_{t \rightarrow \infty} R_S(t) = 0 \quad (4)$$

Failure density $f_S(t)$: The failure density is the density probability function of $F_S(t)$, i.e. the probability that the system fails between t and $t+dt$, is given, for sufficiently small dt 's, by $f_S(t).dt$. Formally,

$$f_S(t) = \frac{d F_S(t)}{dt} \quad (5)$$

Failure rate $r_S(t)$: the failure rate or hazard rate is the probability the system fails for the first time per unit of time at age t . Formally,

$$r_S(t) = \lim_{dt \rightarrow 0} \frac{\Pr\{the\ system\ fails\ between\ t\ and\ t + dt / C\}}{dt} \quad (6)$$

where C denotes the event “the system experienced no failure during the time interval $[0,t]$ ”.

The following properties hold.

$$r_S(t) = \frac{f_S(t)}{R_S(t)} \quad (7)$$

$$R_S(t) = \exp\left[-\int_0^t r_S(u) du\right] \quad (8)$$

VIII.2 Mathematical Definitions of System Availability

Availability $A_S(t)$ and unavailability $Q_S(t)$: the availability of S at t is the probability that S is working at t , given that it all its components were working at 0.

$$A_S(t) = \Pr\{S\ is\ working\ at\ t\} \quad (9)$$

The unavailability is just the opposite.

$$Q_S(t) = 1 - A_S(t) \quad (10)$$

The following properties hold.

$$A_S(t) \geq R_S(t), \text{ for general systems.} \quad (11)$$

$$A_S(t) = R_S(t), \text{ for systems with only non-repairable components.} \quad (12)$$

Conditional failure intensity $\lambda_S(t)$: the conditional failure intensity is the probability that the system fails per unit time at time t , given that it was working at time 0 and is working at time t . Formally,

$$\lambda_S(t) = \lim_{dt \rightarrow 0} \frac{\Pr\{the\ system\ fails\ between\ t\ and\ t + dt / D\}}{dt} \quad (13)$$

where D denotes the event “the system S was working at time 0 and is working at time t ”. The conditional failure intensity is sometimes called Vesely rate. $\lambda(t)$ is an indicator of how the system is likely to fail.

Unconditional failure intensity $w_S(t)$: the unconditional failure intensity is the probability that the system fails per unit of time at time t , given it was working at time 0. Formally,

$$w_S(t) = \lim_{dt \rightarrow 0} \frac{\Pr\{the\ system\ fails\ between\ t\ and\ t + dt / E\}}{dt} \quad (14)$$

where E denotes the event “the system was working at time 0”. This parameter is called “failure frequency” by some authors.

In the case of systems with non-repairable components, the following property holds.

$$w_S(t) = f_S(t), \text{ for systems with only non-repairable components.} \quad (15)$$

In the general case, the following property holds.

$$\lambda_S(t) = \frac{w_S(t)}{A_S(t)} \quad (16)$$

Marginal importance factor of the component c $MIF_{S,c}(t)$: The marginal importance factor is often called Birnbaum importance factor. It can be interpreted, when S is a monotone function, as the conditional probability that, given that c occurred, the system S is failed and c is critical, i.e. a repair of c makes the system working. Formally,

$$MIF_{S,c}(t) = \frac{\partial Q_S(t)}{\partial Q_c(t)} \quad (17)$$

It can be shown that $w_S(t)$ can be evaluated as follows.

$$w_S(t) = \sum_{c \in S} MIF_{S,c}(t) \cdot w_c(t) \quad (18)$$

The unconditional failure intensity of components can be determined from their probability laws. Table 8.1 gives the values computed by Aralia.

<i>Probability Distribution</i>	<i>Unconditional failure intensity</i>
Constant	0
Exponential	$\lambda \times A_c(t)$
Gamma-Lambda-Mu	$\lambda \times A_c(t)$
GLM-asymptotic	0
MTT	$1/MTTR \times A_c(t)$
Weibull	0
Periodic-test (full)	$\lambda \times A_c(t)$
Periodic-test (simple)	$\lambda \times A_c(t)$
Constant Mission Time	0
NRD	0
Dormant	0
Standby	0
Gamma	0
Uniform	0
LogUniform	0
Beta	0
Binomial	0
Chi-Squared	0
Poisson	0
Bound-time	$w_c(t)$ of the modified law
Factor	$f \times w_c(t)$

Table 8.1. Unconditional Failure Intensities

VIII.3 Approximations of the Reliability

If the system S under study is made only of non-repairable components, $r_S(t) = \lambda_S(t)$. In the general case, this equality doesn't hold.

VIII.3.1 Murchland Lower Bound

Let $N_S(t)$ be the number of failures the system experimented between time 0 and t . Let $E[X]$ denote the mathematical expectation of a random variable X . Then, according to Markov inequality, the following property holds.

$$F_S(t) = \Pr(N_S(t) \geq 1) \geq E[N_S(t)]$$

$w_S(t)$ be interpreted as the derivative of $E[N_S(t)]$. Hence, the Murchland lower bound of the reliability.

$$F_S^{[M]} \geq \int_0^t w_S(t).dt$$

Indeed, $F_S^{[M]}(t)$ is close to $F_S(t)$ only for small values of t .

VIII.3.2 Barlow-Proschan lower bound

In [BP76], Barlow and Proschan remark that the Mean Time To Failure (MTTF) is always greater than the Mean Up Time (MUT). They show also the following equality.

$$MUT = \frac{A_S(\infty)}{w_S(\infty)} = \frac{1}{\lambda_S(\infty)}$$

From equality (26), they derive the following upper bound of the unreliability.

$$F_S^{[BP]} \geq t \cdot \lambda_S(\infty)$$

VIII.3.3 Vesely Approximations

The underlying idea of both Vesely approximation of the reliability is to substitute $\lambda_S(t)$ for $r_S(t)$ in equation (8). The full Vesely approximation $F_S^{[V]}(t)$ is defined as follows.

$$F_S^{[V]}(t) = 1 - \exp\left[-\int_0^t \lambda_S(u) du\right]$$

The asymptotic Vesely approximation $F^{[V,\infty]}_S(t)$ is defined as follows.

$$F^{[V,\infty]}_S(t) = 1 - e^{-\lambda_S^{(\infty)} \cdot t}$$

This latter approximation works for large values of t only.

VIII.3.4 Equivalent Lambda

The unconditional failure intensity is sometimes called the “instantaneous equivalent lambda”. In some reliability studies, regulation authorities require to compute its mean value through a period of time. This “mean equivalent lambda” is computed as follows.

$$\lambda_S^{[Mean]}(t) = \frac{\int_0^t \lambda_S(t) \cdot dt}{t}$$

VIII.4 Aralia Commands

Aralia provides the user with commands to compute the following parameters.

- Unconditional Failure Rate, denoted by `UFI`. This parameter is computed according to equation (18).
- Conditional Failure Rate, denoted by `CFI`. This parameter is computed according to equation (16).
- Murchland Lower Bound of the Reliability, denoted by `Fmu`. `Fmu` is actually an approximation of the system unreliability.
- Barlow-Proshan Lower Bound Approximation of the Reliability, denoted by `Fbp`. `Fbp` is actually an approximation of the system unreliability.
- Vesely Asymptotic Approximation of the Reliability, denoted by `Fav`. `Fav` is actually an approximation of the system unreliability.
- Vesely Full Approximation of the Reliability, denoted by `Ffv`. `Ffv` is actually an approximation of the system unreliability.
- Mean Equivalent Lambda, denoted by `ELm`.

The Aralia command to compute these parameters is as follows.

```
compute <reliability-parameter-selector>
  [from <handle-selector>]
  <variable-selector>
  [<time-schedule>] [<tries>] [<doreset>] [<order>]
  [<redirection>] ;
```

The arguments of the above command are as follows.

- `<reliability-parameter-selector>` is a selector of parameters to compute, e.g. `Pr`, `UFI`, ...

- `<handle-selector>` is a selector of data structure handles (i.e. BDD, ZBDD or SoP names) on which the computation is to be performed. By default, the computation is performed on the BDD.
- `<variable-selector>` is a selector of variables for which the computation is to be performed.
- `<time-schedule>` sets the mission times at which the computation is to be performed. There are two ways to set mission times:
 - at `<time1>`, `<time2>`, ... i.e. a list of floating point numbers separated with commas.
 - from `<first-time>` to `<last-time>` step `<time-increment>`
- `<tries>` sets the number of tries (for Monte-Carlo simulations and sensitivity analyses, see chapter IX). The syntax for this option is as follows.


```
tries <number-of-tries>
```
- `<doreset>` if this option is set to 1, parameters of probability laws are no reset to their mean value. The syntax for this option is follows.


```
— reset {0,1}
```
- `<order>` sets the order of the Sylvester-Poincaré development (for SoP only).
- `<redirection>` is a redirection directive to print results into a text file. The syntax for redirection is as follows.


```
> "file-name" /* overwrite the file */
>> "file-name" /* append results to the file */
```

For instance,

```
compute UFI,CFI from BDD,SoP topEvent at 20, 500, 1100 ;
```

VIII.5 Safety Integrity Levels

Aralia provides the user with a command to deal with Safety Integrity Levels. Safety Integrity Levels are defined by the norms IEC 61508 and IEC 61511. They are “a measure of the quality or the dependability of a system which has a safety function”, or in other words, “a measure of the confidence with which the system can be expected to perform that function”. In the cited norms, Safety Integrity Levels are defined differently whether functions are with a low or a high demand rate. In the Aralia context, we consider only functions with a low demand rate. In that case, the Safety Integrity Level L of a system S at time t is derived straight from the unavailability $Q_S(t)$ by the following formula.

$$10^{-(L+1)} \leq Q_S(t) < 10^{-L}$$

The cited norms consider actually levels 1 to 4. It is worth to notice that the level usually depends on the time t , especially if the system embeds periodically tested components. The Aralia command to assess Safety Integrity Levels is as follows.

```
compute SIL
  [from <handle-selector>]
  <variable-selector>
  [<time-schedule>] [<tries>] [<doreset>] [<order>]
  [<redirection>] ;
```

This command computes first (for each selected variable) a curve of the unavailability, by considering the dates given by the time schedule, plus the singular points (obtained from an analysis of periodically tested components), plus a number of intermediate points to smooth the curve and to detect when a threshold is crossed. Then the mean value of the unavailability is computed. Finally, the sojourn times in each Safety Integrity Level are determined.

When a number of tries is given, the printed curve is the one obtained from the default values of parameters. Mean values and standard deviations for sojourn times are given.

VIII.6 Summary

The commands to assess reliability parameters are recalled Table 8.2.

<i>Name</i>	<i>Syntax</i>
UFI CFI Fmu Fbp Fav Ffv ELm	compute <reliability-parameter-selector> [from <handle-selector> <system-selector> [<time-schedule>] [<tries>] [<doreset>] [<order>] [<redirection>] ;

Table 8.2. Commands to compute reliability parameters

The command to assess Safety Integrity Levels is recalled Table 8.3.

<i>Name</i>	<i>Syntax</i>
SIL	compute SIL [from <handle-selector> <system-selector> [<time-schedule>] [<tries>] [<doreset>] [<order>] [<redirection>] ;

Table 8.3. Command to deal with Safety Integrity Levels

IX SENSITIVITY ANALYSES

IX.1 Why to perform sensitivity analyses?

It is often the case that reliability parameters are known only up to a given uncertainty. For instance, the mean time to failure of a component can be suspected to be about 1000 hours, although it may vary significantly around this value. In such cases, it may be interesting to let reliability parameters vary and to observe the incidence of these variations of the final result (e.g. the top event probability). Importance factors provide a mean to observe reliability parameters independently (*mutatis mutandis*). Another good approach consists in performing Monte-Carlo simulations on their values. Aralia offers such a possibility. That's the so-called sensitivity analyses.

IX.2 How to perform sensitivity analyses?

Commands to compute probabilities, importance factors and reliability parameters are described chapters VII and VIII. All of these commands accept the option:

```
tries <number-of-tries>
```

When this option is activated, a Monte-Carlo simulation is performed on the values of parameters. Parameters are drawn according to the distributions described VI. For instance, the command

```
compute Pr topEvent at 100, 1000 tries 1000;
```

performs a Monte-Carlo simulation of 1000 histories on the computation of the probability of `topEvent`. The simulation algorithm is as follows.

```
for try=1 to number-of-tries do
  draw parameters of laws of basic events
  foreach quantity Q to compute do
    foreach mission time t do
      compute Q(t)
    done
  done
done
```

For each computed quantity, it is possible to display the mean value, the standard deviation, the 95% confidence range, the 95% error factor and any number of quantiles.

Recall that the confidence range $[X_{0,05}, X_{0,95}]$ associated with a confidence level of 0.95 is as defined follows.

$$\begin{aligned} X_{0,05} &= \exp[\mu - 1.645\sigma] \\ X_{0,95} &= \exp[\mu + 1.645\sigma] \end{aligned}$$

The 95% error factor EF is defined as follows.

$$EF = \sqrt{\frac{X_{0,95}}{X_{0,05}}} = e^{1.645\sigma}$$

The quantiles are computed on the fly. Therefore, the given values are not the exact one (although they are in general very close to actual values).

The command `set` is used to set the flags that indicate what must be displayed. The flags are the following.

Mean values	<code>option set display-mean-values {on, off};</code>
Standard deviation	<code>option set display-standard-deviations {on, off};</code>
Confidence ranges	<code>option set display-confidence-ranges {on, off};</code>
Error factors	<code>option set display-error-factors {on, off};</code>
Quantiles	<code>option set display-quantiles <integer> ;</code>

For the last command, the parameter is the wanted number of quantiles. Hence, if the given value is 0, no quantile is displayed, if the value is 4, the four quantiles are displayed, if the value is 100, the one hundred percentiles are displayed and so on. E.g.

```
option set display-mean-values on;
option set display-quantiles 10;
compute Pr topEvent at 10, 100, 1000 tries 10000;
```

The values of this option can be obtained via the command `display` (using the same identifiers). E.g.

```
option display display-mean-values;
```

As all other display commands, the above one can be redirected into a file. It is possible to get all options at once by means of the following command.

```
option display interpreter-options;
```

The values of parameters of probability laws, of probabilities of basic events and of the various computed values can be displayed at each try. Commands to set these options are as follows.

```
Parameters    option set display-parameter-values {on, off};
Probabilities option set display-leaves-probabilities {on, off};
Quantities    option set display-tries {on, off};
```

IX.3 The pseudo-random numbers generator

Aralia implements a congruential pseudo-random numbers generator recommended in Numerical Recipes in C:

W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, editors.
Numerical Recipes in C: the Art of Scientific Computing. Cambridge
 University Press, 1988-1992. ISBN 0-521-43108-5.

Congruential generators are based on the following principle: given a multiplier a and a modulus m and an initial value s_0 . Then, the sequence of values is generated according to the following equation:

$$s_{n+1} = a \times s_n \bmod m$$

Since an overflow may result of the multiplication, the modulus is factorized as follows.

$$m = aq + r, \text{ i.e., } q = \lfloor m/a \rfloor \text{ and } r = m \bmod a$$

Then, it can be show that the following equality holds.

$$a \times s \bmod m = \begin{cases} a(s \bmod q) - r \lfloor s/q \rfloor & \text{if it is } \geq 0 \\ a(s \bmod q) - r \lfloor s/q \rfloor + m & \text{otherwise} \end{cases}$$

The Aralia generator takes three parameters (four if one includes the seed): the modulus m , the multiplier a and a mask M (both are integers). The quotient q and the remainder r are computed as indicated above. The next value of the seed is computed as follows.

```
s = s xor M
s = a*(s mod q) - r*(s / q)
if (s<0) s = s+m
```



```
s = s xor M
```

The cited reference recommends the following values for s, a and M.

```
m = 2147483647 (=  $2^{31} - 1$ )
```

```
a = 16807 (=  $7^5$ )
```

```
M = 123459876
```

Two commands make it possible to display and to set the current value of the seed (that may be any integer).

```
display seed [<redirection>;  
set seed <integer>;
```


X SELECTORS

Most of the Aralia commands apply to one or more variables. A variable is uniquely referred by its name. There are however several means to select a variable or a group of variables. Selectors apply not only on logical variables but also on probability law parameters, attributes,... In order to illustrate how these selectors work, we consider throughout this section the following set of equations.

```
r1 := (g1 & g2);  
r2 := (g1 & g3);  
g1 := (e1 | e2);  
g2 := (e2 | e3);  
g3 := (e3 | e4);
```

There are several categories of selectors: basic selectors, set operations, selectors that operate through variable definitions, predicate selectors, sorts and selectors for pattern matching.

X.1 Basic selectors

Basic selectors are names (of variables) and the two constant selectors `{ }` (empty set) and `*` (the full reference set).

For instance, the command `"compute BDD x;"` compute the BDD of the variable `x`, if any. The command `"display parameter *;"` prints all the named probability law parameters.

A selector describes a set of objects (variables, parameters, attributes,...). Hence the selector `x` represents the singleton `{x}` if there is a variable `x` and the empty set otherwise.

X.2 Set operations

Since selectors represent sets, set operations as available as selectors:

`S1, S2, ..., Sn` is the union of selectors `S1, S2, ..., Sn`.

`S1 ^ S2 ^ ... ^ Sn` is the intersection of selectors `S1, S2, ..., Sn`.

$S_1 / S_2 / \dots / S_n$ is the set difference of selectors S_1, S_2, \dots, S_n , i.e. $(S_1/S_2)/S_3 \dots$

Braces '{' and '}' are used as parentheses to resolve priorities.

In our example, the commands "display definition r_1, g_1, g_2 ;" and "display definition $*\{r_1, g_2\}$;" display the definitions of variables r_1, g_2 and g_2 and respectively r_2, g_1 , and g_3 (input variables have no definition).

X.3 Selectors that operate through definitions

The following selectors operate through variable definitions.

children(S): this selects all the variables that occurs immediately under variables selected by S. E.g., **children(r_1)** selects g_1 and g_2 .
children(children(r_2)) selects e_1, e_2 and e_3 .

parents(S) : this is the dual selector of children. E.g., **parents(e_1, g_2)** selects g_1 and r_1 .

leaves(S): this selects all the input variables (basic events) that occur in the definitions of the variables selected by S. E.g. **leaves(g_2, g_3)** selects e_2, e_3 and e_4 .

roots(S): this is the dual selector of leaves. E.g. **roots(e_1)** selects r_1 and r_2 while **roots(e_4)** selects only r_2 .

descendants(S): this selects all the variables that occur in the definition of the variables selected by S. For instance, **descendants(r_1)** selects the set $r_1, g_1, e_1, e_2, g_2, e_3$. By default, descendants are selected in pre-order, as in the previous example. To get them in post-order, it suffices to add the specifier **post-order** as follows. **descendants($S|post-order$)**. For instance, **descendants($r_1|post-order$)** selects the set $e_1, e_2, g_1, e_3, g_2, r_1$.

ancestors(S) : this selects all the variables in the definitions of which a variable selected by S occurs. As previously, a pre-order traversal is performed by default. A post-order traversal is obtained by adding the specifier **post-order**.

Remark: for both descendants and ancestors collector, it is possible to set a maximum level for collecting variables. E.g. **descendants/2(g)** and **ancestors/3(g)** select respectively the children and grandchildren of g and grand-grand-parents, grand-parents and parents of g . Moreover, variables with attribute "always-expanded" are not taken into account to determine the level.

X.4 Predicate selectors

Predicate selectors are of the following form.

```
`[`<boolean-expression>`]`(<selector>)
```

They extract from the set of objects that are selected by their arguments those that verify the predicate. Predicate are Boolean expressions (see chapter XI for a complete description of available expressions).

For instance, `[(:>name $> f) and (:>name $< h)](*)` selects all the variables whose name is lexicographically greater than 'f' and less than h, i.e. in our example, g1, g2, g3.

X.5 Sorts

The sort selector is in the following form.

```
sort `[` {<,>} <field>`]`(<selector>)
```

It sorts items selected by its argument according to the order defined between brackets. Fields are described in chapter XI.

For instance, `sort[$< :>name](*)` applied to variables sorts variables in lexicographic order.

X.6 Pattern-matching

Aralia offers some (limited) pattern-matching possibilities. A pattern-matching selector verifies that the type of the formula (and of its children) matches a given pattern. Its syntax is as follows.

```
`/' <pattern> '/'`(<selector>)
```

Patterns are as follows.

`{.,and,or,not}[(<pattern>)]`: this matches respectively any formula, an and gate, an or gate and a not gate. If the argument is present, this pattern moreover verifies that the children of the formula match it.

`leaf`: this matches an input variable.

`{gates,roots}(<pattern>)`: this matches respectively a gate variable and a root variable. If the argument is present, this verifies that the definition of the gate verifies it.

`<pattern>'|'...'|'<pattern>`: this matches a formula if at least one of the patterns matches it.

`<pattern>'*'`: this matches any number of occurrences of the pattern `<pattern>`. This pattern is used as argument and should be applied to matches children of formula.

`module`: this matches modules. Modules have to be detected first by means of the command `'compute modules;'`.

Patterns are parenthesed by means `'(' and ') '`.

Examples of pattern-matching selectors:

`/leaf/(*)`: selects all leaves.

`/gate(and)/(*)`: selects all the gates in the store whose definition is an and.

`/gate(.leaf*)/(*)`: selects all the gates that are associated with a formula whose children are all leaves.

X.7 Summary

The available selectors are recalled Table 10.1.

Name	Syntax
Full reference set	*
Empty set	<code>'{ '' }'</code>
Name	<code><identifier></code>
Union	<code><selector> [, <selector>]+</code>
Intersection	<code><selector> [^ <selector>]+</code>
Set difference	<code><selector> [/ <selector>]+</code>
Parentheses	<code>'{ ' <selector> ' }'</code>
Children	<code>children(<selector>)</code>
Parents	<code>parents(<selector>)</code>
Leaves	<code>leaves(<selector>)</code>
Roots	<code>roots(<selector>)</code>
Descendants	<code>descendants[/ <integer>](<selector> [{pre-order,post-order}])</code>
Ancestors	<code>ancestors[/ <integer>](<selector> [{pre-order,post-order}])</code>
Predicates	<code>'[' <boolean-expression> ']' (<selector>)</code>
Sorts	<code>sort '[' {<, >} <expression> ']' (<selector>)</code>
Pattern-matching	<code>/ <pattern> / (<selector>)</code>

Table 10.1. Available selectors

Patterns are as recalled table 10.2.

Name	Syntax
------	--------

type matching	{.,leaf,gate,root,and,or,not,module}
Sequence	[(<pattern>)]
Disjunction	<pattern>*
Parenttheses	<pattern> ... <pattern>
	`(` <selector> `)`

Table 10.2. Available patterns

XI EXPRESSIONS, FIELDS AND ATTRIBUTES

XI.1 Expressions

Expressions are used in filters and selectors. Their value depends on the context in which are assessed. The syntax for expression is as follows.

```
<expression>
  ::= <integer> | <float> | <identifier> | <string>
  ::= <field>
  ::= <attribute>
  ::= <boolean-expression>
  ::= #(<boolean-expression> [ , <boolean-expression>]* )
  ::= meet( <selector> )
  ::= ( <expression> )

<boolean-expression>
  ::= not <expression>
  ::= <expression> [ and <expression> ]+
  ::= <expression> [ or <expression> ]+
  ::= <expression> {=, #, <, >, <=, >=} <expression>
```

Fields and attributes are described in the following sections of this chapter.

The expression `#(E1,E2,...,En)` counts the number of (Boolean) expressions among the E_i 's that are satisfied in the current context.

The expression `meet(<selector>)` counts the number of objects selected by the given selector that belong to the current context.

The comparators `=`, `#`, `<`, `>`, `<=` and `>=` compares expression values. If these expressions are numerical, they are interpreted as numbers. If they are symbols, they are interpreted as strings.

XI.2 Fields

Fields are quantities associated with Aralia objects. The general syntax for fields is as follows.

`:> <identifier>`

The interpretation of a field depends on the object to which it refers.

Fields associated with variables and literals :

`:>name` stands for the name of the variable.

`:>bdd-index` stands for the BDD index of the variable (0 if no index is associated with the variable).

`:>sop-index` stands for the SoP index of the variable (0 if no index is associated with the variable).

Fields associated with products.

`:>order` stands for the number of literals in the products.

`:>probability(<mission-time>)` stands for the probability of the product at the given date.

`:>Pr` same as `:>probability`.

`:>rank` stands for the rank, i.e. the order of appearance, of the product.

Fields associated with handles.

`:>encoding` stands for the string that describes the type of the handle (SBDD, ZBDD, SoP, ...).

XI.3 Attributes

Attributes are user defined quantities (strings) associated with variables. The syntax for attributes is as follows.

`::<identifier>`

Commands to manage attributes are as follows.

```
attribute set <attribute-selector> <variable-selector>  
    <expression> ;
```

```
attribute clear <attribute-selector> <variable-selector> ;
```

```
attribute display <attribute-selector> <variable-selector>  
    [<redirection>] ;
```

XII FILTERS

XII.1 What are filters used for?

Filters are used in three circumstances:

1. To display products encoded by a data structure.
2. To compile a data structure (typically a ZBDD) into another (typically a SoP). This compilation is achieved by one of the three commands `compute SBDD`, `compute ZBDD` and `compute SoP`.
3. To compute minimal cutsets with the ZWC algorithm.

Data structures are accessed by their names, so-called handles. The syntax of the above commands is as follows.

```
display {occurrences,orders,product-number,products}
  <handle(s)> <variable(s)>
  [ <filter> ]
  [ <redirection> ];

compute {SBDD,ZBDD,SoP} from <handle> [to <handle>]
  <variable(s)> [ <filter> ];

compute ZWC [from <handle>] [to <handle>]
  <variable(s)> [ <filter> ];
```

A filter selects among the products encoded by the source data structure those that are relevant. In the case of the display products command, filters are also used to tell Aralia what should be displayed in addition to the products (order, probability, contribution, ...).

Here follows two examples of filters. With the first one, Aralia is told to display probabilities and orders of products. With the second one, Aralia is told to select products whose order is less than 3 and whose probability is greater than 1.0e-4.

```
display products ZQC top { display :>Pr, :>order } ;
products(ZQC(top)) {
  display :>Pr, :>order
}
{a, b} 1e-5 2
{a, c, d} 0.001 3
```


end

```
display products ZQC top
{ verify (:>order <= 3) and (:>Pr >= 1.0e-4) } ;
products(ZQC(top)) {
    verify (:>order <= 3) and (:>Pr >= 1.0e-4)
}
{a, c, d}
end
```

XII.2 Syntax of filters

Filters group a number of directives inside braces '{' and '}'. Commands are separated with spaces. Their syntax is as follows.

`verify <Boolean-expression>`

This directive specifies that the selected products must verify the given boolean expression.

`display <expression> [, <expression>]+`

This command applies only for the `display` command. It specifies expressions that are evaluated against each displayed product. The value of these expressions are displayed after the product.

`compute [at <float>] [from <handle>]`

The directives `verify` and `display` may require to compute probabilities and contributions. This directive sets the mission time at which the probabilities and contributions are computed and the data structure from which they are computed.

`keep :>rank <minimum-rank> <maximum-rank>`

`keep :>Pr <number>`

`keep :>order <number>`

In its first form, this directive is used to display products page by page. It tells Aralia to keep only products whose rank, i.e. order of appearance, lies between the given bounds. This cannot be done via the directive `verify` for the rank depends on the satisfaction of the constraint (therefore a constraint such as `:>rank>=10` would be never satisfied).

In its second and third forms, this directive is used to keep the most important products, i.e. respectively the `<number>` products with the highest probabilities and the lowest order. Carefull, if this option is set, the algorithms work in two steps: first they determine a threshold (on order, on probability) and second they compute the quantity of interest with this threshold. This process explains why the number of products that are actually taken into account is not always exactly `<number>`.

The syntax of expressions is described chapter XI. Expressions are built on constants, the usual Boolean and numerical operators, some specific operators, and the notion of field. A field is a quantity associated with the object against which the expression is evaluated. Fields associated with products are as follows.

:>order

This stands for the number of literals in the products.

:>probability(<mission-time>), :>Pr(<mission-time>)

This stands for the probability of the product at the given date.

:>rank

This stands for the rank, i.e. the order of appearance, of the product.

XII.3 Summary

Filters are used to select products in commands `display`, `compute SBDD`, `ZBDD`, `SoP` and `compute ZRC`. The syntax of filters is given Table XII.1.

<pre> <filter> ::= '{' [<compute>] [<verify>] [<display>] [<keep>] '}' <compute> ::= compute [at <float>] [from <handle>] <verify> ::= verify <Boolean-expression> <display> ::= display <expression> [, <expression>]+ <keep> ::= keep :>rank <integer> <integer> ::= keep :>Pr <integer> ::= keep :>order <integer> </pre>

Table XII.1. Syntax of filters.

XIII OPTIONS

This chapter describes some of the options of the Aralia engine.

XIII.1 The command option

A number of options make it possible to tune the Aralia engine. They are set through the command “option”.

```
option clear <option-selector> ;
```

This command resets the selected options to their default value.

```
option display <option-selector> [<redirection>] ;
```

This command displays the values of the selected options.

```
option set <option-selector> <expression> ;
```

This command sets the selected options to the given value.

XIII.2 Available options

XIII.2.1 *Options for sensitivity analyses*

display-confidence-ranges:

To display 95% confidence ranges when performing sensitivity analyses. Value: {on,off}. Default value on.

display-error-factors:

To display the error factor when performing sensitivity analyses. Value: {on,off}. Default value off.

display-leaves-probabilities:

To display the probabilities of basic events (at each try) when performing sensitivity analyses. Value: {on,off}. Default value off.

display-means:

To display mean values when performing sensitivity analyses. Value: {on,off}. Default value on.

display-parameters-values:

To display the values of probability law parameters (at each try) when performing sensitivity analyses. Value: {on,off}. Default value off.

display-standard-deviations:

To display the standard deviations when performing sensitivity analyses. Value: {on,off}. Default value on.

display-tries:

To display the values obtained at each try when performing sensitivity analyses. Value: {on,off}. Default value off.

quantiles:

To display quantiles when performing sensitivity analyses. Value: the number of quantiles. Default value 0.

XIII.2.2 Options for multiple mission times computations

display-time-maximums:

To display the maximum of a value through the time when performing a computation at different mission times. Value: {on,off}. Default value off.

display-time-minimums:

To display the minimum of a value through the time when performing a computation at different mission times. Value: {on,off}. Default value off.

display-time-means:

To display the mean of a value through the time when performing a computation at different mission times. Value: {on,off}. Default value off.

display-time-sums:

To display the sum of values through the time when performing a computation at different mission times. Value: {on,off}. Default value off.

add-singularity-points:

When this option is on, dates at which periodically tested components enter or exit a test period are automatically added to the schedule. Value: {on,off}. Default value off.

XIII.2.3 Other options

prompt1:

The prompt displayed by Aralia when the engine waits for a new command. Value: any string (e.g. "my prompt –"). Default value "aralia >".

prompt2:

The prompt displayed by Aralia when the engine waits the end of a command. Value: any string (e.g. "and ? "). Default value "? ".

trace-file-header:

The header printed by Aralia when the command 'trace' is called.

significant-digits:

The number of significant digits to be given when displaying a floating point number. Value: the number of significant digits. Default value 6.

SP-NEGF:

SP-NEGS:

SP-MiBS:

SP-MaBS:

SP-MiT:

SP-ImTh:

SP-PBEP:

SP-VERB:

Not documented.

verbose:

To make some commands verbose. Value: {on,off}. Default value off.

clear-failed-computations:

To remove all the handles created during a BDD computation that failed (for any reason, memory exhausted, time elapsed or user interruption). When this option is set, the BDD garbage collector is called too. Value: {on,off}. Default value off.

format:

To set the output format. This option is fragile. Available format are 'aralia', 'XML', 'SOP and 'Item'.

XIV GLOSSARY

Binary Decision Diagram (BDD): a BDD is a compact encoding of the truth table of a formula. BDDs are one of the three internal representation of Boolean formulae used in Aralia.

Bound-time (law): one of the probability law available in Aralia.

Cardinality: a cardinality connective is denoted $\#(l, h, [F_1, \dots, F_n])$. It is satisfied if and only if at least l and at most h among the F_i 's are satisfied.

CMT (law): stands for Constant Mission Time. one of the probability law available in Aralia.

Constant (Boolean): either 1 (true) or 0 (false).

Constant (law): one of the probability law available in Aralia.

Constant (parameter): one of the probability law parameter available in Aralia.

Dormant (law): one of the probability law available in Aralia.

Factor (law): one of the probability law available in Aralia.

Exponential (law): one of the probability law available in Aralia.

GLM-asymptotic (law): one of the probability law available in Aralia.

Gate: a gate variable is a variable that occurs as the left member of an equation.

GLM: stands for Gamma-Lambda-Mu. One of the probability law available in Aralia.

Input variable: an input variable (or a leaf) is a variable that does not occur as the left member of an equation. Input variables are terminal events.

Leaf: see input variable.

Literal: a literal is either a variable or its negation.

Lognormal (parameter): one of the probability law parameters available in Aralia.

Normal (parameter): one of the probability law parameters available in Aralia.

NRD (law): one of the probability law available in Aralia.

Output variable: an output variable (or a root) is a variable that does not occur in the right member of an equation. Output variables are top events.

Root: see output variable.

Selector: a selector is a mean provided by Aralia to select variables in a store.

Store: a store is a set of equations.

Sum-Of-Products (SoP): explicit representation of sets of minimal cutsets. SoP are one of the three internal representation of Boolean formulae used in Aralia.

Uniform (parameter): one of the probability law parameters available in Aralia.

Zero-Suppressed Binary Decision Diagram (ZBDD): compact representation of sets of minimal cutsets derived from BDD. ZBDDs are one of the three internal representation of Boolean formulae used in Aralia.

XV SUMMARY OF ARALIA COMMANDS

In what follows, we use the following notations:

- atb-selector stands for attribute-selector
- hdl-selector stands for handle-selector
- prm-selector stands for parameter-selector
- var-selector stands for variable-selector

XV.1 Boolean Equations

```

<store>
  ::= <equation> <store>
  ::=

<equation>
  ::= <variable> := <formula> ;

<formula>
  ::= 0 | 1
  ::= <variable>
  ::= - <formula>
  ::= (<formula> | <formula> [ | <formula> ]+ ])
  ::= (<formula> & <formula> [ & <formula> ]+ ])
  ::= (<formula> = <formula> [ = <formula> ]+ ])
  ::= (<formula> # <formula> [ # <formula> ]+ ])
  ::= (<formula> => <formula>)
  ::= (<formula> <= <formula>)
  ::= (<formula> ? <formula> : <formula>)
  ::= @(<integer> , '['<formula> [, <formula>]+']')
  ::= #(<integer>, <integer> , '['<formula> [, <formula>]+']')
  ::= exists <variable> [, <variable>]* <formula>
  ::= forall <variable> [, <variable>]* <formula>
  ::= cofactor <literal> [, <literal>]* <formula>
  ::= '[' { |, &, =, #} , <var-selector>']'
  ::= '[' @, <integer> , <var-selector>']'
  ::= '[' #, <integer> , <integer> , <var-selector>']'
  ::= '[' exists, <var-selector> , <formula>']'
  ::= '[' forall, <var-selector> , <formula>']'

<variable>
  ::= [a-z,A-Z][a-z,A-Z,0-9,_,-,.]*

<literal>
  ::= <variable> | - <variable>

```

XV.2 Probability distributions

To associate a probability distribution with a basic event, use the following command.

```
basic-event set <var-selector> <parameter> ;
```

The syntax of parameters is as follows.

```
<parameter>
  ::= <float>
  ::= pi
  ::= mission-time
  ::= <identifier>
  ::= ( [ <parameter> and ]+ )
  ::= ( [ <parameter> or ]+ )
  ::= not <parameter>
  ::= <parameter> {=, #, <, >, <=, >=} <parameter>
  ::= - <parameter>
  ::= ( [ <parameter> + ]+ )
  ::= ( [ <parameter> - ]+ )
  ::= ( [ <parameter> * ]+ )
  ::= ( [ <parameter> / ]+ )
  ::= abs( <parameter> )
  ::= {acos, asin, atan, cos, cosh, sin, sinh, tan, tanh}( <parameter> )
  ::= {exp, log, log10}( <parameter> )
  ::= mod( <parameter>, <parameter> )
  ::= pow( <parameter>, <parameter> )
  ::= sqrt( <parameter> )
  ::= {ceil, floor}( <parameter> )
  ::= {min, max, mean}( [ <parameter> , ]+ )
  ::= ite( [ <parameter> , ]3 )
  ::= switch( [ case( <parameter> , <parameter> ) , ]* <parameter> )
  ::= <probability-distribution>
  ::= <random-deviate>
  ::= $<identifier>

<probability-distribution>
  ::= <parameter>
  ::= exponential( [ <parameter> , ]2 ) // lambda mission-time
  ::= exponential( [ <parameter> , ]4 ) // gamma lambda mu mission-time
  ::= GLM( [ <parameter> , ]4 ) // gamma lambda mu mission-time
  ::= Weibull( [ <parameter> , ]4 ) // alpha beta mission-time
  ::= dormant( [ <parameter> , ]3 ) // lambda MTTR delay
  ::= standby( [ <parameter> , ]6 ) // n m r lambda lambda-bar mu
  ::= Dirac( <parameter> ) // delay
  ::= periodic-test( [ <parameter> , ]4 )
  ::= periodic-test( [ <parameter> , ]5 )
  ::= periodic-test( [ <parameter> , ]11 )
  ::= bound-time( <parameter>, <parameter>, <probability-distribution> ) ;

<random-deviate> ::=
  ::= uniform-deviate( [ <parameter> , ]2 ) // minimum maximum
```



```

::= normal-deviate( [ <parameter> , ]2 )      // mean standard-deviation
::= lognormal-deviate( [ <parameter> , ]3 )    // mean err.-factor conf.-level
::= gamma-deviate( [ <parameter> , ]2 )       // k theta
::= beta-deviate( [ <parameter> , ]2 )        // alpha beta
::= histogram(<parameter>, [ bin(<parameter>,<parameter>) , ]+ )

```

XV.3 Command approximate

```

approximate product-number <hdl-selector> <var-selector> [<redirection>] ;
approximate Pr <var-selector> [<mission-times>] [<redirection>] ;

<mission-times>
  ::= at <float> [, <float>]*
  ::= from <float> to <float> step <float>

<redirection>
  ::= > "<file-name>"
  ::= >> "<file-name>"

```

XV.4 Command attribute

```

attribute clear <atb-selector> <var-selector> ;
attribute compute <heuristic> <var-selector> ;
attribute display <atb-selector> <var-selector> [<redirection>] ;
attribute set <atb-selector> <var-selector> <expression> ;
attribute values <atb-selector> <var-selector> [<redirection>] ;

```

XV.5 Command basic-event

```

basic-event set <var-selector> <probability-distribution> ;
basic-event clear <var-selector> ;
basic-event display <var-selector> [<redirection>] ;

BEP {<identifier>, <integer>} <float> ;
BER {<identifier>, <integer>} <float> ;

```

XV.6 Command bdd-order

```

bdd-order clear <var-selector> ;
bdd-order DFLM <var-selector> ;
bdd-order display <var-selector> [<redirection>] ;
bdd-order {move,swap} {up,down} <var-selector> ;
bdd-order round-robin ;
bdd-order set <identifier> <integer> ;
bdd-order sift <var-selector> ;
bdd-order sifting ;

```

XV.7 Command clear

```
clear all ;
clear bdd-unique-table ;
clear handle <hdl-selector> <var-selector> ;
```

XV.8 Command coalesce

```
coalesce <var-selector> ;
```

XV.9 Command compute

```
compute BDD [to <handle>] <var-selector> ;
compute p-BDD [truncated <integer>] [to <handle>] <var-selector> ;
compute p-BDD [! <integer>] [to <handle>] <var-selector>;
compute back from <ZBDD-handle> [to <handle>] <var-selector> ;
compute BDA [from <BDD-handle>] [to <handle>] <var-selector> ;
compute SBDD from <handle> [to <handle>] <var-selector> [<filter>];
compute ZBDD from <handle> [to <handle>] <var-selector> [<filter>];
compute SoP from <handle> [to <handle>] <var-selector> [<filter>];

compute ZPI [! <order>] [from <BDD-handle>] [to <handle>] <var-selector> ;
compute ZPJ [! <order>] [from <BDD-handle>] [to <handle>] <var-selector> ;
compute ZMC [! <order>] [from <BDD-handle>] [to <handle>] <var-selector> ;
compute ZPC [! <order>] [from <BDD-handle>] [to <handle>] <var-selector> ;
compute ZQC [! <order>] [from <BDD-handle>] [to <handle>] <var-selector> ;

compute MCS <MCS-algorithm> [! <order>] [from <BDD-handle>] [to <handle>]
    <var-selector> <var-selector> ;

compute MOCUS [to <handle>] <MOCUS-filter> <var-selector> ;

compute ZMCS-FDT [from <handle>] [to <handle>] <var-selector> [<filter>];
compute ZMCS-FOT [from <handle>] [to <handle>] <var-selector> [<filter>];
compute ZMCS-FWT [from <handle>] [to <handle>] <var-selector> [<filter>];

compute SoP from <handle> [to <handle>] <var-selector> [<filter>];

compute <Pr-selector> [from <hdl-selector>] <var-selector>
    [<mission-times>] [<tries>] [<doreset>] [<order>] [<redirection>] ;
compute <IF-selector> [from <handle-selector>] <var-selector> <var-selector>
    [<mission-times>] [<tries>] [<doreset>] [<order>] [<redirection>] ;
compute <reliability-parameter-selector>
    [from <hdl-selector>] <var-selector>
    [<mission-times>] [<tries>] [<doreset>] [<order>] [<redirection>] ;

<mission-times>
    ::= at <float> [, <float>]*
    ::= from <float> to <float> step <float>
<tries>
    ::= tries <integer>
<doreset>
```

```

::= reset {0,1}
<order>
::= order <integer>

<Pr>
::= { Pr, PP }

<IF>
::= { CPr, CQr, MIF, CIF, DIF, RAW, RRW }

<reliability-parameter>
::= { UFI, CFI, Fmu, Fbp, Fav, Ffv, ELm }

<redirection>
::= > "<file-name>"
::= >> "<file-name>"

```

XV.10 Command display

```

display bdd-unique-table active-nodes-number [<redirection>] ;
display bdd-unique-table free-nodes-number [<redirection>] ;
display bdd-unique-table page-number [<redirection>] ;
display bdd-unique-table default-page-size [<redirection>] ;
display bdd-unique-table maximum-page-number [<redirection>] ;
display bdd-hashtables minimum-size [<redirection>] ;
display bdd-hashtables maximum-size [<redirection>] ;
display bdd-hashtables enlargement-ratio [<redirection>] ;
display bdd-hashtables reduction-ratio [<redirection>] ;
display bdd-entry-table size [<redirection>] ;
display bdd-hashcache minimum-size [<redirection>] ;
display bdd-hashcache maximum-size [<redirection>] ;
display bdd-hashcache enlargement-ratio [<redirection>] ;
display bdd-hashcache reduction-ratio [<redirection>] ;
display bdd-hashcache size [<redirection>] ;
display bdd-hashcache active-entries-number [<redirection>] ;
display bdd-garbage-collection period [<redirection>] ;
display bdd-garbage-collection number [<redirection>] ;
display bdd-reordering strategy [<redirection>] ;
display bdd-reordering threshold [<redirection>] ;
display bdd-reordering period [<redirection>] ;
display sift maximum-growth [<redirection>] ;
display sifting minimum-ratio [<redirection>] ;
display round-robin minimum-improvement [<redirection>] ;
display bdd-statistics [<redirection>] ;

display seed [<redirection>] ;
display interpreter-options [<redirection>] ;
display version [<redirection>] ;

display handle <hdl-selector> <var-selector> [<redirection>] ;
display definition <var-selector> [<redirection>] ;
display variable <var-selector> [<redirection>] ;

display bdd-nodes <hdl-selector> <var-selector> [<redirection>];
display maximum-order <hdl-selector> <var-selector> [<redirection>];
display occurrences <hdl-selector> <var-selector> <filter> [<redirection>];

```



```
display orders <hdl-selector> <var-selector> <filter> [<redirection>];  
display product-number <hdl-selector> <var-selector> <filter> [<redirection>];  
display products <hdl-selector> <var-selector> <filter> [<redirection>];  
display size <hdl-selector> <var-selector> [<redirection>];
```

XV.11 Command echo

```
echo "<string>" ;
```

XV.12 Command exit

```
exit ;
```

XV.13 Command group

```
group set <identifier> <var-selector> order ;  
group set <identifier> <var-selector> <group-type>  
  [ [ <parameter> , ]+ ] <probability-distribution> ;  
group clear <group-selector> ;  
group display <group-selector> [ <redirection> ] ;  
group expand <group-selector> ;  
group sort {>,<} <expression> <group-selector> ;  
  
<group-type> ::= alpha-factor | beta-factor | MGL | phi-factor
```

XV.14 Command help

```
help [<command> [<option>]] ;
```

XV.15 Command history

```
history;  
history <integer> ;
```

XV.16 Command load

```
load "<file-name>" ;
```

XV.17 Command normalize

```
normalize constants <var-selector> ;
```



```
normalize definition <var-selector> ;  
normalize dominated-occurrences <var-selector> ;  
normalize names [<identifier>]4 [<redirection>];
```

XV.18 Command option

```
option clear <option-selector> ;  
option display <option-selector> [<redirection>] ;  
option set <option-selector> <expression> ;
```

XV.19 Command parameter

```
parameter set <prm-selector> <parameter> ;  
parameter clear <prm-selector> ;  
parameter display definition <prm-selector> [<redirection>] ;  
parameter display value <prm-selector> [<redirection>] ;  
parameter rename <identifier> <identifier> ;  
parameter reset <prm-selector> ;  
parameter draw <prm-selector> ;
```

XV.20 Command prune

```
prune <var-selector> <var-selector>;
```

XV.21 Command remove

```
remove <var-selector> ;
```

XV.22 Command rename

```
rename <old-name> <new-name> ;
```

XV.23 Command rename

```
save "<file-name>" ;
```

XV.24 Command rewrite

```
rewrite <heuristic> <var-selector> ;
```

XV.25 Command set

```

set bdd-entry-table size <integer> ;
set bdd-garbage-collection period <integer> ;
set bdd-hashcache size <integer> ;
set bdd-hashcache maximum-size <integer> ;
set bdd-hashcache minimum-size <integer> ;
set bdd-hashcache enlargement-ratio <float> ;
set bdd-hashcache reduction-ratio <float> ;
set bdd-hashtables maximum-size <integer> ;
set bdd-hashtables minimum-size <integer> ;
set bdd-hashtables enlargement-ratio <float> ;
set bdd-hashtables reduction-ratio <float> ;
set bdd-reordering strategy {off,sifting,round-robin} ;
set bdd-reordering threshold <integer> ;
set bdd-reordering period <integer> ;
set bdd-unique-table default-page-size <integer> ;
set bdd-unique-table maximum-page-number <integer> ;
set sift maximum-growth <float> ;
set sifting minimum-ratio <float> ;
set round-robin minimum-improvement <float> ;

set delay-before-interruption <expression> ;
set seed <expression> ;

set from <handle> [<cutoff>] [to <handle>] <var-selector> ;
set key <string> ;
set history {on,off} ;
set $<identifier> <expression> ;

```

XV.26 Command sop-order

```

sop-order clear <var-selector> ;
sop-order DFLM <var-selector> ;
sop-order display <var-selector> [<redirection>] ;
sop-order set <identifier> <integer> ;

```

XV.27 Command sort

```

sort {literals,products} <sop-comparator> <hle-selector> <var-selector> ;
sort children <order> [<iteration-directive>] <var-selector> ;
sort sibship <order> <var-selector> ;

<sop-comparator>      ::= {<,}> <field> [, <sop-comparator>]
<order>               ::= {<,}> <expression> [, <order>]
<iteration-directive> ::= iterate <integer>

```

XV.28 Command store

```

store import <store-selector> ;

```



```
store export <store-selector> ;
store new <identifier> ;
store print <store-selector> ;
store prune <identifier> <identifier>;
store rename <identifier> ;
store remove <store-selector> ;
store reset <store-selector> ;
store set <identifier> ;
```

XV.29 Command store-order

```
store-order clear <var-selector> ;
store-order DFLM <var-selector> ;
store-order display <var-selector> [<redirection>] ;
store-order set <identifier> <integer> ;
```

XV.30 Command system

```
system "<command>" ;
```

XV.31 Command timer

```
timer new <identifier> ;
timer remove <identifier> ;
timer reset <identifier> ;
timer start <identifier> ;
timer stop <identifier> ;
timer restart <identifier> ;
timer print <identifier> [<redirection>] ;
```

XV.32 Command trace

```
trace on "<file-name>" ;
trace off ;
```

XV.33 Command user-order

```
user-order clear <identifier> <var-selector> ;
user-order DFLM pre-order <identifier> <var-selector> ;
user-order DFLM post-order <identifier> <var-selector> ;
user-order DFLM asap-order <identifier> <var-selector> ;
user-order display <identifier> <var-selector> [<redirection>] ;
user-order linear-arrangement <identifier> <var-selector> [<redirection>] ;
user-order set <identifier> <identifier> <integer> ;
```

XV.34 Instructions

```

<instruction>
  ::= <built-in>
  ::= if <expression> <instruction> [ else <instruction> ] fi
  ::= while <expression> <instruction>
  ::= foreach <shell-variable> in <selector> <instruction>
  ::= { <instruction>+ }

<built-in>
  ::= all commands

```

XV.35 Selectors

```

<selector>
  ::= <identifier>
  ::= '{''}'
  ::= *
  ::= '{' <selector> '}'
  ::= <selector> , <selector> [ , <selector> ]
  ::= <selector> ^ <selector> [ ^ <selector> ]
  ::= <selector> / <selector> [ / <selector> ]
  ::= children(<selector>) | parents(<selector>)
  ::= ancestors(<selector> [ '|' {pre-order,post-order} ])
  ::= descendants(<selector> [ '|' {pre-order,post-order} ])
  ::= roots(<selector>) | leaves(<selector>)
  ::= modules(<selector>)
  ::= in:<handle-name>(<selector>)
  ::= '[' <boolean-expression> ']'(<selector>)
  ::= sort '[' {<,,>} <field> ']'(<selector>)
  ::= /<pattern>/ (<selector>)

<pattern>
  ::= '.' [ ( <pattern> ) ]
  ::= <pattern> '*'
  ::= <pattern> [ '|' <pattern> ]+
  ::= leaf | module | {gate,root} [ ( <pattern> ) ]
  ::= {and,or,not} [ ( <pattern> ) ]
  ::= ( <pattern> )

```

XV.36 Filters

```

<filter>
  ::= '{' [<compute>] [<verify>] [<display>] [<keep>] '}'

<compute>
  ::= compute [at <float>] [from <handle>]

<verify>
  ::= verify <Boolean-expression>

```



```

<display>
  ::= display <expression> [, <expression>]+

<keep>
  ::= keep :>rank <integer> <integer>
  ::= keep :>Pr <integer>
  ::= keep :>order <integer>

```

XV.37 Expressions, Attributes, Fields

```

<expression>
  ::= <integer> | <float> | <identifier> | <string>
  ::= <field>
  ::= <attribute>
  ::= <boolean-expression>
  ::= #(<boolean-expression> [ , <boolean-expression>]* )
  ::= meet( <selector> )
  ::= ( <expression> )

<boolean-expression>
  ::= not <expression>
  ::= <expression> [ and <expression> ]+
  ::= <expression> [ or <expression> ]+
  ::= <expression> {=, #, <, >, <=, >=} <expression>

<attribute>
  ::= ::<identifier>

<field>
  ::= :>name
  ::= :>bdd-index
  ::= :>sop-index
  ::= :>probability(<mission-time>)
  ::= :>order
  ::= :>rank
  ::= :>encoding

```