

# AltaRica Checker Handbook

---

user-guide for ARC 1.7  
25 November 2024



A. Griffault, G. Point, A. Vincent

---

This document is the manual for the ARC tool version 1.7.  
Permission is granted to copy and/or distribute this document.

LaBRI - CNRS UMR 5800 - Université Bordeaux  
351, cours de la Libération  
F-33405 TALENCE CEDEX  
FRANCE

# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Where to get ARC ?	1
1.2	Where to send bug reports, comments or requests for new features ?	1
<b>2</b>	<b>ARC at a glance</b>	<b>3</b>
2.1	The game	3
2.2	The model	3
2.3	Getting started with ARC	4
2.4	First computations	5
2.5	Computing the winning strategy	7
<b>3</b>	<b>The arc command</b>	<b>11</b>
3.1	Interactions with ARC	11
3.2	General purpose commands	12
3.2.1	apropos	12
3.2.2	cd	12
3.2.3	echo	13
3.2.4	eval	13
3.2.5	exit	14
3.2.6	gc	14
3.2.7	help	14
3.2.8	info	14
3.2.9	list	15
3.2.10	load	16
3.2.11	pwd	16
3.2.12	remove	17
3.2.13	set	17
3.2.14	show	18
3.2.15	timer	19
3.3	Commands related to AltaRica nodes	19
3.3.1	ca	19
3.3.2	depgraph	20
3.3.3	flatten	20
3.3.4	node-info	21
3.3.5	obfuscate	22
3.3.6	solve	22
3.3.7	stepper	23
3.3.8	target-reduction	24
3.3.9	to-lustre	26
3.3.10	validate	27
3.3.11	chkctl	28
3.4	Commands related to computations using exhaustive engine	28
3.4.1	ts	28
3.4.2	ts-marks	29
3.4.3	show-ts-marks	29
3.5	Commands related to MEC 5 relations	29
3.5.1	card	29

3.5.2	check-card .....	30
3.5.3	pick .....	31
3.5.4	store .....	31
3.6	Computation of sequences and fault trees: cuts and sequences .....	33
3.7	Stochastic simulation: sas .....	35
3.8	Experimental commands .....	37
3.8.1	diag .....	37
3.8.2	sat .....	37
<b>4</b>	<b>Using the ACHECK specifications .....</b>	<b>39</b>
4.1	Overview .....	39
4.2	Representation of the semantics .....	40
4.3	Computing properties of nodes .....	40
4.3.1	Built-in sets .....	40
4.3.1.1	Sets of configurations .....	41
4.3.1.2	Sets of transitions .....	41
4.3.2	Operators .....	42
4.3.3	Using CTL* logic .....	44
4.4	Commands .....	45
<b>5</b>	<b>Using the MEC 5 specifications .....</b>	<b>49</b>
5.1	Writing MEC 5 predicates .....	49
5.2	Built-in MEC 5 relations .....	51
<b>6</b>	<b>Stochastic simulation .....</b>	<b>55</b>
6.1	Stochastically Timed ALTARICA .....	55
6.2	Pre-requisites on models .....	56
6.3	Syntax of extern clauses .....	56
6.4	Clauses for stochastic simulation .....	57
6.4.1	Parameters .....	57
6.4.2	Laws .....	58
6.4.3	Observers .....	59
6.4.4	Memorization of delays .....	59
6.4.5	Priority .....	59
6.4.6	Random choices .....	59
6.5	Example .....	60
<b>7</b>	<b>Altarica Studio .....</b>	<b>63</b>
7.1	Validation tools .....	63
7.2	Simulator .....	66
<b>8</b>	<b>References .....</b>	<b>69</b>
<b>Appendix A</b>	<b>User preferences .....</b>	<b>71</b>
A.1	Shell .....	71
A.2	Acheck .....	72
A.3	Mec V .....	72
A.4	Translation of ALTARICA models into LUSTRE programs .....	72
A.5	Translation of LUSTRE programs into ALTARICA models .....	74

<b>Appendix B</b>	<b>Probabilistic laws</b>	<b>79</b>
B.1	Dirac's law	79
B.2	Empiric law	79
B.3	Erlang's law	79
B.4	Generalized Erlang's law	80
B.5	Exponential law	80
B.6	Exponential law + Wait On Weather delays	80
B.7	Instants Provided in Advance	81
B.8	Instants Fixed in Advance	81
B.9	Log Normal law	81
B.10	Optional laws	82
B.11	Triangular law	82
B.12	Uniform law	83
B.13	Weibull's law	83
B.14	Truncated Weibull's law	84

# 1 Introduction

ARC is a toolbox for the ALTARICA language ([agp99], page 69). The main purpose of ARC is the *model-checking* of systems described with ALTARICA but its aim is to gather several tools for the analysis or compilation of ALTARICA models. ARC gathers works realized on two suites of tools: ACHECK and MEC 5 ([av03], page 69). The current version offers the following features:

- The original ALTARICA language has been extended with:
  - compound types (arrays, structures).
  - the definition of abstract types and signatures of functions (not supported by the model-checking engine).
- ACHECK specifications are supported using explicit or symbolic representation of the state-spaces (however not all the set of commands are supported in both encoding).
- CTL\* formulae can be used in place of fixed-point equations.
- MEC 5 ([av03], page 69) specifications are also supported and are handled using Decision Diagrams (those used within the TOUPIE tool [cr97], page 69). ARC extends the set of predefined sets with, for instance,  $N!reach$  that specifies the set of reachable configurations of the node  $N$ .
- ARC package integrates a small GUI hymbly called ALTARICA STUDIO that implements a graphical simulator.
- Large relations can be serialized in binary files for future use.
- Preprocessors can be specified using the configuration file of ARC.
- Translators for the LUSTRE language (see [gp06], page 69).
- Several algorithms have been implemented to generate sets of sequences or fault trees according to an unexpected configuration [akpv11], page 69.
- A simulator for models decorated with stochastic informations; it permits to evaluate some measure such like MTTF (*mean time to failure*).

## 1.1 Where to get ARC ?

The ALTARICA website, <http://altarica.labri.fr>, housed at LaBRI gathers many informations related to the ALTARICA language. In particular you will find how to download the ARC tarball or the URL of the repository that contains its source code.

## 1.2 Where to send bug reports, comments or requests for new features ?

Bug reports, comments or any new feature requests can be sent to [altarica@groupe.renater.fr](mailto:altarica@groupe.renater.fr). You can also use the contact form of the *Support* section at ALTARICA website.



## 2 ARC at a glance

In this chapter we present a small session with ARC. We do not present all features of the tool; we refer the reader to next chapters for details. Here we simply show the basics of commands and interactions with ARC while studying a game between two opponents. We use ARC to compute a winning strategy for one of the two players.

### 2.1 The game

We consider a game with two players, say *A* and *B*. Player *A* has four coins and each coin has two different sides. Player *A* arranges coins as a square and chooses which side of each coin is visible. *B* never sees the current side of coins. Each turn of the play, *B* requests *A* to change the side of either

- one coin,
- one column or one line of coins,
- or one diagonal.

Each request can be satisfied in different ways by *A*; for instance, when *B* says “one coin”, *A* has four possibilities. Since *B* does not see coins, he can not know the choice made by its opponent. *B* wins the game if all coins are on the same side. Of course *A* has to make his possible to prevent *B* to reach one of the two winning situations. We assume that *A* is honest enough to end the game when he loses.

Our goal in this chapter is to use ARC to find a strategy for player *B* that is winning from all configurations of the game.

### 2.2 The model

The board game (i.e. the coins arranged as a square) is easily described with an ALTARICA node.

- First we define a domain `SIDE` for values that model sides of coins; Booleans could be a good choice but integers 0 and 1 ease the description of properties.
- Then we declare a node `Board` where:
  - the four coins are modeled with a two-dimensional array `c` of values in `SIDE`.
  - the three requests of the player *B* are modeled with three events called `one`, `col_or_line` and `diagonal`.
  - the response of *A* is modeled using transitions. The non-deterministic choices made by *A* is straightforward since ALTARICA allows non-determinism on event occurrences.

We store the ALTARICA model of the game into a file called `game.alt`. The content of the file is given below. In the following lines `$` denotes the prompt of the shell program (e.g. `bash`).

```
$ cat game.alt
domain SIDE = [0,1];

#define FLIP(i,j) c[i][j] := ((c[i][j] + 1) mod 2)

node Board
state c : SIDE[2][2];
event one, col_or_line, diagonal;
trans
  true |- one -> FLIP(0, 0);
  true |- one -> FLIP(0, 1);
  true |- one -> FLIP(1, 0);
  true |- one -> FLIP(1, 1);
```



```

true |- col_or_line -> FLIP(0, 0), FLIP(1, 0);
true |- col_or_line -> FLIP(0, 1), FLIP(1, 1);
true |- col_or_line -> FLIP(0, 0), FLIP(0, 1);
true |- col_or_line -> FLIP(1, 0), FLIP(1, 1);

true |- diagonal -> FLIP(0, 0), FLIP(1, 1);
true |- diagonal -> FLIP(0, 1), FLIP(1, 0);
edon
$

```

A reader used to ALTARICA language should have noticed that the third line of the file is not an ALTARICA sentence. Indeed, it is the definition of a macro-function `FLIP(i, j)` for the C preprocessor. This macro-function is a shortcut used in transitions to flip the coin at line *i* and column *j*. We come back on preprocessing in the next section.

## 2.3 Getting started with ARC

Now that we have written the model of the game we just have to start ARC and load the file `game.alt`. ARC accepts several options followed by filenames (see [Chapter 3 \[The ARC command\]](#), page 11, for details on options). We start the program with the file `game.alt` as argument (the ellipsis below replaces the banner of ARC). Inputs of the user are written using a typewriter font and ARC outputs are printed using *italic* font.

```

$ arc game.alt
...
Loading file 'game.alt'.
game.alt:3: error: syntax error on symbol '#' (any text mode).
arc>

```

Oops! Things does not start very well because the ALTARICA parser finds an error in the file. The error occurs on the first character of the C preprocessor line that defines the macro-function `FLIP(i, j)`.

By default preprocessing of files is disabled; this yields to a syntax error on line 3. To enable it we use the command `set` (see [\[set command\]](#), page 17), to specify in the preference variable `arc.shell.preprocessor.default.command` which preprocessor we want to use by default (here `cpp`):

```

arc>set arc.shell.preprocessor.default.command "/usr/bin/cpp"
arc>

```

Note that the value of preferences are lost between sessions unless the user requests ARC to store them into its configuration file (i.e. `~/arcrc`). To save preferences, use the option `-save` of the `set` command. (See [Appendix A \[User preferences\]](#), page 71, for existing customization variables.)

Now we can reload `game.alt`. If we display the node `Board` (using [\[show command\]](#), page 18) we can see substitutions realized by the preprocessor:

```

arc>load game.alt
Loading file 'game.alt'.
arc>show Board
node Board
state
/* 1 */ c : [0, 1][2][2];
event
/* 1 */ '$';
/* 2 */ diagonal;
/* 3 */ col_or_line;
/* 4 */ one;
trans
true |- one -> 'c[0][0]' := ('c[0][0]'+1) mod 2;
true |- one -> 'c[0][1]' := ('c[0][1]'+1) mod 2;
true |- one -> 'c[1][0]' := ('c[1][0]'+1) mod 2;
true |- one -> 'c[1][1]' := ('c[1][1]'+1) mod 2;

```

```

true |- col_or_line -> 'c[1][0]' := ('c[1][0]'+1) mod 2, 'c[0][0]' := ('c[0][0]'+1) mod 2;
true |- col_or_line -> 'c[1][1]' := ('c[1][1]'+1) mod 2, 'c[0][1]' := ('c[0][1]'+1) mod 2;
true |- col_or_line -> 'c[0][1]' := ('c[0][1]'+1) mod 2, 'c[0][0]' := ('c[0][0]'+1) mod 2;
true |- col_or_line -> 'c[1][1]' := ('c[1][1]'+1) mod 2, 'c[1][0]' := ('c[1][0]'+1) mod 2;
true |- diagonal -> 'c[1][1]' := ('c[1][1]'+1) mod 2, 'c[0][0]' := ('c[0][0]'+1) mod 2;
true |- diagonal -> 'c[1][0]' := ('c[1][0]'+1) mod 2, 'c[0][1]' := ('c[0][1]'+1) mod 2;
true |- '$' -> ;
// assertion is (implicitly) 'true'.
// no initial assignment is specified.
// no initial constraint is specified.
edon
arc>

```

One can notice the presence of an additional event '\$' and a transition `true |-$ ->`. This is the well-known  $\epsilon$  event and its associated transition that are added implicitly by the ALTARICA semantics.

## 2.4 First computations

Our goal is to compute a strategy that is winning from all positions of coins. The size of the state-space of the game (16 configurations) is reasonable enough to be displayed. To do this we use ACHECK commands (see [Chapter 4 \[Using the Acheck specifications\], page 39](#)). Rather than creating a new file containing these commands we use [\[eval command\], page 13](#), that redirects the standard input of the program to the ALTARICA parser. EOF is used to close ACHECK mode.

```

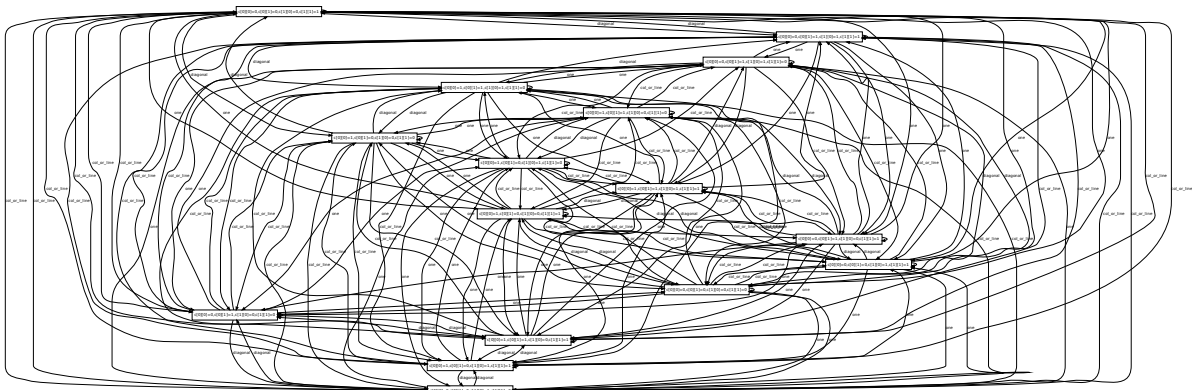
arc>eval
eval>with Board do
  dot(any_s, any_t) > "game.dot";
done
EOF
arc>

```

The `with ... do ... done` sentence specifies that ARC has to apply commands listed between the `do` and `done` keywords to each nodes listed between `with` and `do`. Before executing commands ARC computes the semantics of the node and then applies commands to the semantics.

Here we apply to `Board` the command `dot` (see [Section 4.4 \[Commands\], page 45](#)). This latter displays the state graph of the node using GRAPHVIZ file format (i.e. for the `dot` tool - [\[dot\], page 69](#)). The two keywords `any_s` and `any_t` simply said that all states and all transitions have to be displayed (sometimes it is interesting to restrict these two sets e.g. for counterexamples).

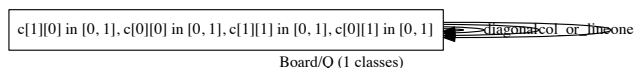
The output of the command is redirected into a file called `game.dot`. Using the DOT tool of GRAPHVIZ we obtain the following graph:



Despite its small size this graph is not very instructive. ARC possesses another command, called `quot`, that outputs the state-graph (yet in DOT format) but this time, states are gathered according to the greatest auto-bisimulation that respects currently computed properties.

```
arc>eval
eval>with Board do
  quot() > "game_q0.dot";
done
EOF
arc>
```

As shown on the following figure the generated graph is not more interesting than the previous one. Up to now no properties have been computed and since all events are possible from all states the greatest auto-bisimulation is the identity.



In order to refine the result we compute small properties that will label states:

- $O$  is the set of states where only one coin is either 0 or 1;
- $CL$  is the set of states where coins form columns or lines;
- $D$  is the set of states where coins form diagonals;
- and finally,  $W$  is the set of states that are winning for  $B$ .

These properties are simply written “ $X := \phi$ ” where  $X$  is the name of the set and  $\phi$  is an ACHECK formula. In our example properties are related to values of coins. When we have to talk about the values of variables one writes the corresponding Boolean expression between square-brackets. For instance, winning states (the set  $W$ ) are those for which all coins display the side 0 or all display the side 1; the corresponding formula for this set can be that the sum of values that label coins is either 0 or 4.

```
arc>eval
with Board do
  W := [c[0][0] + c[0][1] + c[1][0] + c[1][1] = 0 or
        c[0][0] + c[0][1] + c[1][0] + c[1][1] = 4];

  O := [c[0][0] + c[0][1] + c[1][0] + c[1][1] = 1 or
        c[0][0] + c[0][1] + c[1][0] + c[1][1] = 3];

  CL := [c[0][0] + c[1][0] = 2 and c[0][1] + c[1][1] = 0 or
         c[0][0] + c[1][0] = 0 and c[0][1] + c[1][1] = 2 or
         c[0][0] + c[0][1] = 2 and c[1][0] + c[1][1] = 0 or
         c[0][0] + c[0][1] = 0 and c[1][0] + c[1][1] = 2];

  D := [c[0][0] + c[1][1] = 2 and c[1][0] + c[0][1] = 0 or
        c[0][0] + c[1][1] = 0 and c[1][0] + c[0][1] = 2];

  show (all);
  quot() > "game_q1.dot";
done
EOF
/*
 * Properties for node : Board
 * # state properties : 5
 *
 * CL = 4
 * D = 2
 * O = 8
 * W = 2
 * any_s = 16
 *
 * # trans properties : 1
```

```

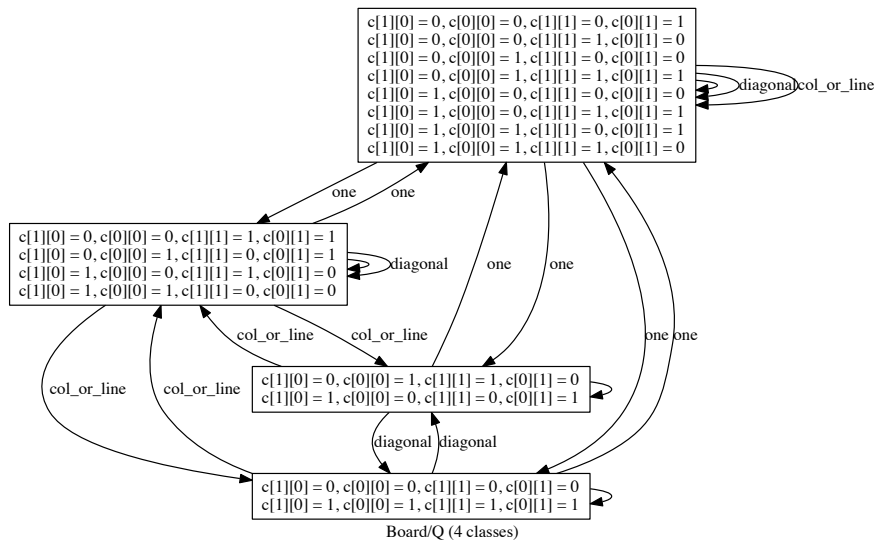
*
* any_t = 176
*/
arc>

```

Before the output of the `quot` command into the file `game_q1.dot` we call the command `show` that displays the cardinalities of currently computed properties. The parameter `all` indicates that all known properties have to be displayed. This keyword can be replaced by a comma-separated list of property identifiers.

The reader can notice that some properties that was not specified have been computed. In fact, several properties are pre-defined in ACHECK (see Section 4.3.1 [Built-in sets], page 40) and computed if necessary (e.g. `any_s`).

The new graph obtained with the `quot` command is displayed below.



This graph shows that the game would have been modeled with only four configurations. Indeed sides of coins (0 and 1) play symmetrical roles and only arrangements of coins (i.e. lines, columns, ...) are relevant rather than their actual position.

Even if the graph is simpler than the previous one, the winning strategy remains unobvious<sup>1</sup>.

## 2.5 Computing the winning strategy

To check the existence of the winning strategy we use MEC 5 specifications (see Chapter 5 [Using the Mec 5 specifications], page 49). The logic of MEC 5 permits to easily write formulas that talk about individual configurations and events rather than sets of configurations and transitions as in the ACHECK language.

MEC 5 specifications are essentially a list of relations (or predicates) definitions. The first one we define is the unary relation that contains all configurations that are winning for  $B$ .

```

arc>eval
eval>Win(s : Board!c) :=
  s.c[0][0] + s.c[1][0] + s.c[0][1] + s.c[1][1] = 0 or
  s.c[0][0] + s.c[1][0] + s.c[0][1] + s.c[1][1] = 4;
EOF
Win (s : Board!c) : 2 elements / 7 nodes
arc>

```

<sup>1</sup> However, with a little effort it is easy to get it!

This declaration starts with the signature of the relation : its identifier (`Win`) and the name and type of each columns (here `s` of type `Board!c`). `Board!c` is the type of configurations of the node `Board`. The prototype is then followed by a first order formula whose models are elements of the defined relation. The formula can refer to the relation itself (i.e. the relation is defined recursively) in which case its models are elements of the underlying fixed-point. We refer the reader to [Chapter 5 \[Using the Mec 5 specifications\], page 49](#), for details on the syntax and semantics of the MEC 5 logic. Note that, since version 1.6, relations computed using ACHECK logic are available in Mec 5 formulas using ! notation; for instance, here we could use `Board!W` instead of defining the relation `Win`.

If ARC has not been started in *quiet* mode (i.e. with the `-q` option) it displays the cardinality of the relation once its computation is done. To display the number of elements of a relation, use [\[card command\], page 29](#):

```
arc>card Win
card (Win) = 2
arc>
```

MEC 5 specifications does not permit to talk about sequences of arbitrary length. To compute the result we enumerate possible lengths of strategies until we found at least one.

We first define a relation  $WSeq_k$  ( $k > 0$ ) with  $k+1$  arguments : a configuration  $s$  and  $k$  events  $e_i$  for  $i = 1, \dots, k$ . A vector  $\langle s, e_1, \dots, e_k \rangle$  belongs to  $WSeq_k$  if and only if the sequence  $e_1, \dots, e_k$  is executed from the configuration  $s$  and the game passes through a winning configuration:

- For  $k = 1$ , the relation is simply:

```
WSeq1(s : Board!c, e1 : Board!ev) :=
  Win(s) or
  [t : Board!c] (Board!t (s,e1,t) => Win (t));
```

- For any  $k > 1$ , the relation is defined using  $WSeq_{k-1}$ :

```
WSeqk
(s : Board!c, e1 : Board!ev, ..., ek
 : Board!ev) :=
  Win(s) or
  [t : Board!c] (Board!t (s, e1, t) => WSeqk-1
  (t, e2, ..., ek
  ));
```

The relation  $S_k$  that contains winning strategy of length  $k$  is a  $k$ -ary relation whose arguments are events  $e_i$  for  $i = 1, \dots, k$ . Its definition simply expresses that the sequence of events is winning from all configurations:

```
Sk
(e1 : Board!ev, ..., ek
 : Board!ev) :=
  [s : Board!c] WSeqk
  (s, e1, ..., ek
  );
```

These relations,  $WSeq_k$  and  $S_k$ , have been written until  $S_k$  becomes non-empty; we gathered them into a file `game.mec5`. The first non-empty relation is  $S_7$  and is a singleton:

```
arc>load game.mec5
Loading file 'game.mec5'.
card (Win) = 2
card (S1) = 0
card (S2) = 0
card (S3) = 0
card (S4) = 0
card (S5) = 0
card (S6) = 0
card (S7) = 1
arc>show S7
e1. = diagonal, e2. = col_or_line, e3. = diagonal, e4. = one, e5. = diagonal,
```

```
e6. = col_or_line, e7. = diagonal  
arc>
```

The reader can use the step-by-step simulator of ALTARICA STUDIO (see [Section 7.2 \[Simulator\]](#), [page 66](#)) to try to win against the strategy given by the relation  $S7$  ...



## 3 The `arc` command

ARC has been designed as a command-line tool like shell programs used on Unix-like systems (e.g. `bash` or `csch`). If you execute the ARC command, the program simply displays a welcome banner and the prompt `arc>`:

```
$ arc
arc 1.7.0
Copyright (c) 2002-2018 LaBRI - CNRS & University of Bordeaux.
All rights reserved.
This software is distributed under AltaRica Public License (see COPYING file).

Please report any bug to altarica[at]groupes.renater.fr
or on project webpage http://altarica.labri.fr/

arc>
```

The user can add arguments to the ARC command. An argument that starts with a dash (-) is considered as an option; an argument that is not an option is considered as a filename. The current version of ARC allows the following options:

- b            This option indicates that ARC is used in batch mode i.e. it exits after the interpretation of all its arguments (files and options).
- c *script*   This option indicates to ARC that the next argument, *script*, should be interpreted as a list of `arc` commands.
- d            This option exists only for debugging purposes; it enables the display of many ugly messages.
- h or --help       Display the general syntax of ARC command-line arguments and the list of allowed options.
- q            This option specifies that ARC should be run in a *quiet* mode i.e. eliminating verbose informing messages. In particular this option disables the display of the “Welcome” banner.
- x            This option enables enable command/response mode. This mode is used by GUI AltaRica studio. It implements a small query/response protocol.
- V or --version    This option displays the current version of ARC. If -q is not specified it prints out compilation options and many other data related to the current version in use.

### 3.1 Interactions with ARC

Depending of the installation host the ARC prompt might be compiled using the READLINE library ([[rl](#)], [page 69](#)) or not. This library is now used by many prompt-based programs (e.g. Unix shells) to ease interactions with the user.

The READLINE library memorizes commands executed by the user. When ARC exits this *history* is stored into a file named by default `~/.arc_history` but it can be changed using user preferences (see [Appendix A \[User preferences\]](#), [page 71](#)). The next time ARC is started interactively, this file is reloaded and the user gets back commands of its previous session.

Each command is essentially the identifier of the command followed by its arguments. A semi-colon (;) can be used to separate several commands entered on the same command line. For instance, below three commands are executed from the same command line: the first one defines a new relation `R`, the second command displays the cardinality of `R` and the last one outputs elements of `R`.



```
arc>eval EOF; card R ; show R
eval>R(i : [0,10]) := <n : [0,10]>(i = 3 * n);
eval>EOF
R (i : [0, 10]) : 4 elements / 2 nodes
card (R) = 4
i = 0
i = 3
i = 6
i = 9
```

The reader familiar with previous version of ARC should have noticed that `eval` is used in a different way (see [\[eval command\]](#), page 13).

Outputs of commands can be redirected into a file or into a pipe to another program. This mechanism uses the same notation than the one used by Unix shells:

- `cmd arg1 ... argn > filename` outputs the results of the command `cmd` into the file called `filename`. If `filename` does not exist it is created; else, it is made empty and overwritten.
- `cmd arg1 ... argn >> filename` outputs the results of the command `cmd` at the end of the file called `filename`. If `filename` does not exist it is created.
- `cmd arg1 ... argn | "program progarg1 ... progargn"` outputs the results of the command `cmd` to the standard input of `program` executed with arguments `progargi`. For instance

```
apropos preferences | more
```

should display the list of user preferences using the pager command `more`.

## 3.2 General purpose commands

### 3.2.1 apropos

Looks for a keyword in ARC internal documentation.

#### Synopsis:

```
apropos keyword
```

#### Description:

This command looks for a *keyword* in ARC internal documentation and lists all the available topics related to it. Topics can then be consulted using the [\[help command\]](#), page 14.

*Example :*

```
arc>apropos apropos
Topics related to 'apropos':
help - Display help informations about given topics.
apropos - Looks for a keyword in ARC internal documentation.
set - Lists/sets/saves/loads preferences.
commands - Command list
pages - List of help pages
```

### 3.2.2 cd

Change current working directory.

#### Synopsis:

```
cd path
```

#### Description:

This command simply changes the current working directory of ARC to *path*. This directory can be displayed using [\[pwd command\]](#), page 16.

### 3.2.3 echo

Write its arguments on standard output.

#### Synopsis:

```
echo arg1 ... argn
```

#### Description:

Print arguments on standard output; arguments are separated by space characters.

*Example* :

```
arc>echo "This is a test"
This is a test
arc>
```

### 3.2.4 eval

Interpret given arguments or standard input as it would be by the load command.

#### Synopsis:

`eval text1 text2 ...` : Interprets each *text*<sub>1</sub> *text*<sub>2</sub> ...

`eval` : Reads and interprets text from the standard input until the word EOF is read at the begin of the line or an end of file character e.g. *CTRL-D*).

`eval txt`: Reads and interprets text from the standard input until the word *txt* is read at the begin of the line or an end of file character e.g. *CTRL-D*).

#### Description:

This command interprets strings given as arguments or the characters typed on the standard input. Strings might be either an ALTARICA text, MEC 5 equations or ACHECK commands. If no argument is passed to the command then the standard input stream is interpreted.

*Example 1*: Definition of an integer constant. ARC rejects redefinition of constants.

```
arc>eval
eval>const N : integer = 3;
eval>EOF
arc>eval
eval>const N : integer = 3;
eval>EOF
<eval>:1: error: redefinition of constant 'N'.
arc>
arc>show N
const N : integer = 3;
arc>
```

*Example 2*: Evaluation of a MEC 5 predicate where the constant N is used (*^D* means *CTRL-D*).

```
arc>eval
eval> R(x : [0,10]) := <y : [0,10]> (x = N * y);
eval> ^D
R (x : [0, 10]) : 4 elements
arc>show R
R contains :
x = 0
x = 3
x = 6
x = 9

arc>
```

### 3.2.5 `exit`

Quit the ARC program.

**Synopsis:**

`exit`

`quit`

Terminates the ARC session.

**Description:**

This command requests ARC for normal termination. Modified preferences are not saved when the program terminates. If you want to keep your settings for next sessions use [\[set command\]](#), [page 17](#) with the `-save` option.

### 3.2.6 `gc`

Calls garbage collector of the Decision Diagram engine.

**Synopsis:**

`gc`

**Description:**

This command explicitly calls the garbage collector of DDs.

### 3.2.7 `help`

Display help about ARC commands or syntax.

**Synopsis:**

`help` : With no argument the command is equivalent to 'help help'

`help topic` : Displays informations related to *topic*

**Description:**

This command is the companion of [\[apropos command\]](#), [page 12](#) in the internal documentation of ARC. This command displays help informations about a given *topic*. Use command `apropos` to look for topics related to some keyword.

You can also type `help pages` to get a list of all available help pages.

For long help messages you should pipe the help command with a shell command like `more` or `less`.

### 3.2.8 `info`

Display detailed informations related to specified objects.

**Synopsis:**

`info D id1 id2 ... :`

Displays information about objects identified by *id<sub>1</sub>*, *id<sub>2</sub>*, ... in dictionary *D* where *D* is one of the following name: `constants`, `domains`, `nodes`, `signatures`, `relations`, `dd-assignments`, `dd-all-paths`, or `order`.

**Description:**

For each given identifier, the command displays informative data related to the object. ARC looks for identifiers into its table *D*; a prefix can be used to specify the table e.g `const` for `constants`. If no identifier is given, the command displays informations about all known objects.

Available dictionaries/tables are the following:

- constants**  
The command returns informations related to declared (global) constants of the model.
- domains**  
The command returns informations related to declared (global) domains of the model.
- dd-all-paths**  
Identifiers are names for MEC 5 relations. The command display the DD encoding the relations in DOT format. All paths from the root to leaves are displayed.
- dd-assignments**  
Identifiers are names for MEC 5 relations. The command display the DD encoding the relations in DOT format. Only paths from the root to the positive leaf  $\top$  are displayed.
- nodes**  
For each specified ALTARICA node, the command displays its different dimensions (i.e. its number of variables, events, ...) and its hierarchy. Since they are not instantiated, no information is displayed for templates.
- order**  
For specified MEC 5 relations, the command gives the order on variables used to compute its decision diagram.
- relations**  
Identifiers are names of relations. For each one, the command displays the signature of the relation, its size (i.e., its number of elements) and the number of nodes of its underlying decision diagram.
- signatures**  
The command recalls the prototype of the specified signatures.

*Example* : Below we create a new relation `R`. The command `info` displays the signature of the relation, its cardinality and the size of the data structure (i.e. the number of nodes of the DD) used to store it.

```
arc>list all
arc>eval
eval>R(x : [1,4]) += x <= 2;
eval>EOF
R (x : [1, 4]) : 2 elements
arc>list all
defined relations : R
arc>info rel R
R : [1, 4] -> bool
cardinality       : 2
data structure size : 2
arc>
```

See also [\[list command\]](#), page 15.

### 3.2.9 list

Lists existing objects.

#### Synopsis:

- `list kinds` : Gives the list of supported sets of objects.
- `list all` : List all existing objects gathered by kinds.
- `list kindname` : List all objects of kind 'kindname'

#### Description:

This command gives the list of identifiers of existing objects: `constants`, `domains`, `nodes`, `root-nodes`, `signatures`, `relations`, `commands`.

To display a specific object, use the `[show command]`, page 18 or try `[info command]`, page 14.

```
arc>list all
arc>eval
const NB_ELEMENTS = 3;
const MIN_VALUE = 1;
const MAX_VALUE = MIN_VALUE + NB_ELEMENTS - 1;
domain R = [MIN_VALUE, MAX_VALUE];

rel(s : R, t : R) := s = t;
eval>EOF
rel (s : [1, 3], t : [1, 3]) : 3 elements / 5 nodes
arc>list all
defined constants : MAX_VALUE, MIN_VALUE, NB_ELEMENTS
defined domains : R
defined relations : rel
arc>
```

### 3.2.10 load

Load the specified files.

#### Synopsis:

```
load file1 file2 ...
```

Open and interpret files *file<sub>1</sub>*, *file<sub>2</sub>*, ... All files are processed even if an error occurs in one of them.

#### Description:

This command is used to load into memory ALTARICA models, ACHECK or MEC 5 specifications, LUSTRE programs (some restriction are applied, see `[gp06]`, page 69 for details), DD encoded relations (see `[store command]`, page 31) or ARC scripts files. The parser is selected according to the file extension.

- If the file terminates with `.arc` then it is interpreted as an ARC script.
- If the file terminates with `.lus` then it is interpreted as a Lustreprogram and is automatically translated into AltaRica.
- If the file terminates with `.rel` then it is interpreted as a serialized DD encoding a relation.
- In other cases the file is passed to the parser used for ALTARICA, ACHECK, MEC 5 and ARC scripts (these languages share the same parser).

For the latter case, the user can specify a preprocessing tool; see `[arc.shell.preprocessor]`, page 71.

### 3.2.11 pwd

Display current working directory.

#### Synopsis:

```
pwd
```

#### Description:

This command outputs the current working directory from where ARC executes its commands. To change it `[cd command]`, page 12 must be used.

### 3.2.12 remove

Remove resources allocated to specified objects.

#### Synopsis:

```
remove D id1 id2 ...
```

Remove objects identified by *id<sub>1</sub>*, *id<sub>2</sub>*, ... in dictionary *D*.

```
remove D all
```

Remove all objects in dictionary *D*.

```
remove all
```

Remove all objects in all dictionaries.

#### Description:

For each given identifier, the command releases resources used by the associated object. Identifiers are related to some namespace or dictionary belonging to the following list: **constants**, **domains**, **nodes**, **signatures** or **relations**. Prefix of these names can be used.

*Example :*

```
arc>list relations
defined relations :
arc>eval
eval>R(x : [1,4]) += x <= 2;
eval>EOF
R ( x : [1, 4] ) : 2 elements / 2 nodes
arc>list rel
defined relations : R
arc>remove relation R
arc>list rel
defined relations :
```

### 3.2.13 set

Lists/sets/saves/loads preferences.

#### Synopsis:

**set** : With no argument the command lists all existing preferences with their current value.

**set id**: Displays the current value of the preference named *id*.

**set id value**: Changes the current value of preference named *id*.

**set -save**: Requests ARC to store all preferences into the configuration file (*~/arcrc*).

**set -load**: Forces ARC to reload preferences from its configuration file. Table of preferences is not reset.

#### Description:

The behavior of ARC's commands is influenced by some preferences (or customization options) that the user can modify. This preferences are simply couples (*name,value*). The **set** command permits to change or to display the value assigned to such options. Modified preferences are not kept between ARC sessions. To be persistent changes must be explicitly stored into the configuration file *~/arcrc*. The values are saved using the option **-save**. During a session, values stored into the configuration file might be reloaded using the **-load** option.

*Example :* The following examples changes the value of **arc.shell.prompt.0** that defines the prompt of ARC. Then ARC is restarted and one can see that the normal prompt is back.

```

$ arc -q
arc>set arc.shell.prompt.0
arc.shell.prompt.0 = arc>
arc>set arc.shell.prompt.0 "bow> "
bow> set arc.shell.prompt.0
arc.shell.prompt.0 = bow>
bow> ^D
$ arc -q
arc>set arc.shell.prompt.0
arc.shell.prompt.0 = arc>

```

Using `-save` we store the new prompt into the configuration file.

```

arc>set arc.shell.prompt.0 "bow>"
bow>set -save
bow>^D
$ cat ~/.arcrc | grep prompt.0
arc.shell.prompt.0 = bow>
$ arc -q
bow>

```

To get informations about existing preferences and their possible values, type `help preferences` and/or pipe the command with an *ad hoc* call to `grep` command; you can also try [\[apropos command\]](#), page 12.

*Example :*

```

arc>help preferences | "grep timers"
acheck.timers :
    Enable/disable timers for acheck computations.
mec5.timers :
    Enable/disable timers for MEC 5 computations.
    Enable/disable the display of timers for acheck computations
arc>apropos timers
Topics related to 'timers':
preferences - User preferences
cuts - Computation of sets of scenarios
set - Lists/sets/saves/loads preferences.
timer - Measurement of execution times for ARC commands
a2l-preferences - Options for Altarica-To-Lustre translator
arc>

```

### 3.2.14 show

Display specified objects.

#### Synopsis:

```
show  $id_1, \dots$ 
```

Display objects identified by  $id_i$

#### Description:

The command looks for an object (relation, signature, ...) with identifier  $id_i$  and displays it. If several objects have the same identifier then all are displayed.

*Example :*

```

arc>eval
eval>R(x : [1,4]) += x <= 2;
eval>EOF
R (x : [1, 4]) : 2 elements
arc>list relations
defined relations : R
arc>show R
x in [1, 2]
arc>

```

### 3.2.15 timer

Timers to measure CPU time.

#### Synopsis:

**timer**: Lists existing timers

**timer start** *timer-id*: Starts the timer *timer-id*; if it does not exist it is created.

**timer reset** *timer-id*: Stop and reset accumulated time for the timer *timer-id*.

**timer stop** *timer-id*: Stops timer *timer-id*. Elapsed CPU time since last **start** is accumulated.

**timer elapsed** *timer-id*: Displays current elapsed CPU time since last **start** of timer *timer-id*.

**timer print** *timer-id*: If timer *timer-id* is stopped, displays accumulated time since creation or last **reset** of the timer.

**timer remove** *timer-id*: Releases resources allocated to and unregisters timer *timer-id*.

#### Description:

This command permits to create timers. Timers measure CPU time and not calendar time. Each one is identified by a name specified at creation and recalled each time a value of the timer is displayed. Two values are associated to each timer:

1. the current elapsed CPU time since the last **start** command;
2. the total amount of CPU time accumulated since the last **reset** or since the creation of the timer.

A timer is created and started using the **start** command. The identifier specified at creation is reused when values of the timer are displayed; if the name of the timer is not a simple word then it must be enclosed between double-quotes (""). **stop** subcommand permits to temporarily suspend the timer and to store time elapsed since **start** into the accumulator; the value of this accumulator can be displayed using **print**. **elapsed** permits to display elapsed time since the last **start** without stopping the timer. Finally **reset** subcommand stops the timer and reset its accumulator to 0.

*Example :*

```
arc>timer start "My Timer"
arc>eval
eval>R(s : [0,100], t : [0,100]) := s = 2*t;
eval>R (s : [0, 100], t : [0, 100]) : 51 elements / 53 nodes
arc>timer elapsed "My Timer"
elapsed (My Timer) = 0.030
arc>timer
My Timer
arc>timer remove "My Timer"
arc>timer
arc>
```

## 3.3 Commands related to AltaRica nodes

### 3.3.1 ca

Compute and display the constraint automaton (i.e. its semantics) of the given node.

#### Synopsis:

**ca** [--mec] *node-id*<sub>1</sub> ... *node-id*<sub>*n*</sub> : Displays constraint automata of nodes *node-id*<sub>*i*</sub>

#### Description:

This command computes and displays the constraint automaton that corresponds to the flat semantics of the node identified by *node-id*<sub>*i*</sub>. The main difference between



the commands `ca` and `flatten` is the removal of compound types (i.e. structures and arrays) and quantified formulas in the constraint automaton.

If the option `--mec` is specified, dot notation (`.`) used to separate fields of a structure is replaced by `^`.

See also [\[flatten command\]](#), page 20.

### 3.3.2 depgraph

Displays informations related to dependency graph of a constraint automaton.

#### Synopsis:

```
depgraph [--option]* node-id
```

where *option* can be: `constraints`, `dot-func-deps`, `func-deps`, `no-dep-graph`, `trans-assert`, `trans-classes`, `with-events` or `--with-func-deps`.

#### Description:

This command displays informations about existing dependencies between objects that compose the constraint automaton of node *node-id*. By default the command outputs dependency graph between all objects except events (i.e., assertions, transitions and variables) in DOT file format.

Several options can be specified:

- . `--constraints`: display assertions that have not been converted into functional dependencies;
- . `--dot-func-deps`: display functional dependencies in DOT file format;
- . `--func-deps`: display discovered functional dependencies;
- . `--no-dep-graph`: disable the output of the whole dependency graph;
- . `--trans-assert`: displays for each macro-transitions the list of assertions that influence its enabling condition;
- . `--trans-classes`: displays for strongly-connected component of the dependency graph restricted to transitions;
- . `--with-events`: add events to dependency graph
- . `--with-func-deps`: replace assertions with discovered functional dependencies.

### 3.3.3 flatten

Compute and display flat semantics of given nodes.

#### Synopsis:

```
flatten id1 id2 ...
```

Display, for each *id<sub>1</sub>*, *id<sub>2</sub>*, ..., the flat nodes semantically equivalent to their semantics.

#### Description:

This command computes the flat semantics of given ALTARICA nodes. The flat semantics is a node equivalent to the original one but without any hierarchy nor priorities or broadcast vectors.

As shown on the following example, `flatten` works with models that use abstract data types and signatures. This is not the case of [\[ca command\]](#), page 19.

*Example :*

```
arc>eval
sort Queue;
const EMPTY : Queue;
const FULL : Queue;

sig put : Queue * integer -> Queue;
```

```

sig remove : Queue -> Queue;
sig first  : Queue -> integer;

node Consumer
  flow i : integer;
      o : integer;
  state q : Queue; init q := EMPTY;
  state s : integer init s := 0;

  event put > consum;
  trans
    q != EMPTY |- consum -> q := remove (q), s := first (q);
    q != FULL  |- put   -> q:= put (q, i);
  assert
    o = s;
edon
EOF
arc>flatten Consumer
// flat semantics of node Consumer
node Consumer
  flow
    i : integer;
    o : integer;
  state
    q : Queue;
    s : integer;
  event
    'consum';
    'put';
    'consum' < 'put';
  trans
    q!=EMPTY and not (q!=FULL) |- 'consum' -> q := remove(q), s := first(q);
    q!=FULL  |- 'put'   -> q := put(q,i);
  assert
    o=s;
  init
    q := EMPTY, s := 0;
  // no initial constraint is specified.
edon

```

See also [gp00], page 69 and [ca command], page 19.

### 3.3.4 node-info

Get details about components of nodes.

#### Synopsis:

`node-info node-id subcommand`

Outputs data related to node *node-id* according to *subcommand* (or one of its non-empty prefixes) where *subcommand* can be:

`children` Lists sub-nodes of the node

`events` Lists events of constraint automaton *node-id*.

`flow-variables`

Lists flow variables of constraint automaton *node-id*.

`max-conf-card`

Returns the maximal cardinality of the set of configurations (i.e., the size of the Cartesian product of domains) of constraint automaton *node-id*.

`state-variables`

Lists state variables of constraint automaton *node-id*.

**transitions**

Lists macro-transitions of constraint automaton *node-id*.

**varorder** Displays the order used by decision diagram package for variables of constraint automaton *node-id*

**Description:**

This command returns data about a given node or its equivalent constraint automaton. This command is mainly used by external scripts (e.g ALTARICA STUDIO).

### 3.3.5 obfuscate

Obfuscate ALTARICA files.

**Synopsis:**

`obfuscate [options] file1 file2 ...`

where options are:

`--algo=id` specify the algorithm used to obfuscate identifiers.

`--dump-table=filename` dump into a file the translation table (a SED script) from identifiers to obfuscated identifiers.

`--dump-rev-table=filename` dump into a file the translation table (a SED script) from obfuscated identifiers to actual identifiers.

**Description:**

This command renames all identifiers of given files *file<sub>s</sub>* to make them anonymous even if they can be exploited using ARC.

Currently only one algorithm is implemented (`--algo=sequential`). It consists in the replacement of each identifier of the file by a number preceded by an underscore character. Each replacement is stored into a dictionary and all occurrences of an identifier are replaced using the same number. All files passed as arguments to the command share the same dictionary; this way the coherence between files is ensured.

Note that files that contains ARC scripts are not obfuscated.

### 3.3.6 solve

Solve the global assertion of the given nodes.

**Synopsis:**

`solve node-id`

`glucose node-id`

Outputs assignments that satisfy assertions of node *node-id*.

`solve node-id max`

`glucose node-id max`

Outputs at most *max* assignments that satisfy assertions of node *node-id*.

`solve node-id max expr`

`glucose node-id max expr`

Outputs at most *max* assignments that satisfy Boolean expression *expr* over variables of node *node-id*.

**Description:**

This command has been added mainly for debugging purposes. Its is used to compute the configurations authorized by assertions of a node (and its underlying hierarchy). The set of solutions of assertions is represented explicitly; thus this command should

be use carefully. The number of displayed solutions can be limited using the second argument *max*; if *max* is negative no limit is applied.

A third argument can be a Boolean expression that is solved in place of assertions; this expression is built over variables of *node-id*.

The Glucose solver ([GLU], page 69) can be called in place of the internal solver using the command `glucose`.

*Example* : The following example models a pipe that can be broken. When the failure occurs the pipe becomes leaky and its output rate becomes non deterministically inferior to its input rate.

```
$ cat leaky-pipe.alt
const MAX_RATE = 5;

node Pipe
  flow in, out : [0, MAX_RATE];
  state mode : { nominal, degraded };
  init mode := nominal;
  event failure;
  trans mode = nominal |- failure -> mode := degraded;
  assert
    case {
      mode = nominal : out = in,
      else out < in
    };
edon

node Main
  sub p : Pipe;
  assert p.in = MAX_RATE;
edon

$ arc -qb leaky-pipe.alt -c 'solve Main'
// solutions to constraints of node 'Main' (solver=solve)
p.mode = degraded, p.in = 5, p.out = 0
p.mode = degraded, p.in = 5, p.out = 1
p.mode = degraded, p.in = 5, p.out = 2
p.mode = degraded, p.in = 5, p.out = 3
p.mode = degraded, p.in = 5, p.out = 4
p.mode = nominal, p.in = 5, p.out = 5
$
```

### 3.3.7 stepper

A textual step-by-step simulator.

#### Synopsis:

`stepper node-id start [form1] ... [formn]` : Solves configurations given as constraints.

`stepper node-id valid-trans [form1] ... [formn]` : Lists transitions that can be triggered from a given configuration.

`stepper node-id trigger Tindex [form1] ... [formn]` : Trigger a transition from a given configuration.

`stepper node-id vars` : Lists the variables of the node used by the stepper.

#### Description:

This command is a textual step-by-step simulator. It has been designed for ALTARICA STUDIO GUI and not to be used directly by a user. The first argument is always the identifier of a node. It is then followed by a subcommand name:

- **start**: Without argument, the command returns the initial configuration(s). Else, arguments are interpreted as formulas over variables of the node. Then, the com-

mand returns configurations that satisfy both the conjunction of  $form_i$ s and assertions of the node. The result of the command is a list of configurations (one per line). Each configuration is a comma-separated list of equalities  $varname=value$  where  $varname$  is a variable of the node and  $value$  its assigned value in the configuration.

- **valid-trans**: This subcommand returns indices of macro-transitions that can be triggered from the given configuration. The conjunction of formulas  $form_i$  is interpreted as the current configuration of the node. The semantics of  $form_i$ s in conjunction with assertions must be a singleton. The command returns a list of transitions with indices which are used next with the **trigger** command.
- **trigger**: The subcommand computes configurations that are reached from the given configuration when transition with index  $Tindex$  is triggered. As for previous commands  $form_i$ s are interpreted as a set of configurations that must be a singleton. The command displays a list of configurations.
- **vars**: variables (of the node) used by the stepper are displayed one variable per line.

### 3.3.8 target-reduction

Project an ALTARICA node on the cone of influence (COI) of a given formula.

#### Synopsis:

```
target-reduction [--option=filename]* nodeid formula
```

where *option* can be: `depgraph-before`, `depgraph-after`, `goal-before`, `goal-after`, `fdep-before`, `fdep-after`, `fdep-dot-before`, `fdep-dot-after`, `validate`.

#### Description:

This command syntactically reduces ALTARICA node specified by *nodeid* according to the cone of influence induced by *formula*. *formula* is any Boolean formula over variables of *nodeid* as can be specified in ACHECK specifications (see [Chapter 4 \[Using the Acheck specifications\]](#), page 39). The resulting node contains sufficient parts of the original one to study accessibility of states satisfying *formula*.

Options are used to get informations related to reduction process. A filename can be passed as argument to each option. The output generated by the option is printed into the specified file; if the filename is empty the output is redirected onto standart output.

Except for **validate**, options are suffixed either by **before** or by **after**. The suffix indicates when the option is applied i.e. *before* or *after* the reduction; both suffixes can be used.

The meaning of options is the following:

- **depgraph**: displays the dependency graph (using DOT format) of elements that compose the node: variables, events, assertions, transitions.
- **goal**: displays the formula used to guide the reduction. After reduction *formula* can have been rewritten according to discovered functional dependencies.
- **fdep**: displays dependencies between variables of the nodes.
- **fdep-dot**: same as **fdep** except that the result is printed using DOT file format.
- **validate**: apply [\[validate command\]](#), page 27 to the reduced node.

The following example models  $N$  components (`Cell`) that form a kind of pipeline; the output of the first one goes into the input of the second one, the output of the second goes into the input of the third one, and so on. The ALTARICA code is the following (we use PHP preprocessor):

```

$ cat pipeline.alt.php
const N=<?php echo $argv[1] ?>;

node Cell
  flow i, o : bool;
  state s : bool;
  assert o = (if s then i else false);
  event act;
  trans true |- act -> s := not s;
edon

node Pipeline
  sub c : Cell[N];
  assert
  <?php for($i = 0; $i < $argv[1] - 1; $i++) { ?>
    c[<?php echo $i ?>].o = c[<?php echo $i+1 ?>].i;
  <?php } ?>
edon

```

Now imagine we want to observe the output of the  $i^{\text{th}}$  component of the pipeline; say the third one (i.e. `c[2]`). Actually, even if ALTARICA semantics does not define any notion of orientation of flow variables, the semantics of *this* model is such that variables of component `c[i]` only depend on variables of components `c[0]`,  $\dots$ , `c[i - 1]`.

The following script shows the use of `target-reduction` for the observation of `c[2].o`. The resulting *flat* node is reduced to elements that belong to components that influence the value of `c[2].o`.

```

$ cat pipeline.arc
set arc.shell.preprocessor.php.args 5
load pipeline.alt.php
target-reduction --goal-after=pipeline.goal Pipeline "c[2].o"

$ arc -qb pipeline.arc
// statistics:
// number of variables : 4
//   flow variables : 1
//   state variables : 3
// max cardinality : 2
// number of events : 4
// number of assertions : 0
// number of transitions : 4
node Pipeline
  flow // 1 flow variables
    'c[0].i' : bool;
  state // 3 state variables
    'c[0].s' : bool;
    'c[1].s' : bool;
    'c[2].s' : bool;
  event // 4 events
    'c[0].act';
    'c[1].act';
    'c[2].act';
  trans
    true |- 'c[2].act' -> 'c[2].s' := not 'c[2].s';
    true |- 'c[1].act' -> 'c[1].s' := not 'c[1].s';
    true |- 'c[0].act' -> 'c[0].s' := not 'c[0].s';
edon

```

One can remark that it remains only one flow variable `c[0].i` and, furthermore, `c[2].o`, the observed variable has disappeared. In order to reduce the number of variables, several ones are replaced by discovered functional dependencies. These substitution are applied to the input formula. The option `--goal-after=` of `target-reduction` (see above) outputs the resulting rewritten formula:

```

$ cat pipeline.goal

```

```
'c[0].s' and 'c[0].i' and 'c[2].s' and 'c[1].s'
```

### 3.3.9 to-lustre

Translate given nodes into a LUSTRE program.

#### Synopsis:

```
to-lustre id1 id2 ...
```

Translate nodes *id*<sub>1</sub>, *id*<sub>2</sub>, ... into a LUSTRE program.

#### Description:

The command translates each node *id*<sub>*i*</sub> into a *equivalent* LUSTRE node. The translation is done on all required items i.e. domains, sub-nodes, ... The translation algorithm and its restrictions are described in details in [gp06], page 69.

The translation is parameterized by several preferences (see [Translation of ALTARICA models into LUSTRE programs], page 72 or simply type `help a2l-preferences`).

*Example* : The following example gives the translation of an ALTARICA Switch node into LUSTRE.

```
$ cat switch.alt
node Switch
  flow
    i : bool : in;
    o : bool : out;
  event
    open, close;
  state
    is_open : bool;
  init
    is_open := true;
  assert
    is_open => (i = o);
  trans
    is_open |- close -> is_open := false;
    not is_open |- close -> is_open := true;
edon
$ arc -qb switch.alt -c 'to-lustre Switch'
--
-- TRANSLATED NODES
--

-- WARNING : output/local variable 'f_o' in node 'Switch' is not defined.
node Switch(close_, f_i, open_ : bool)
  returns (f_o : bool);
var
  ec_0, ec_1, ec_3, s_is_open : bool;
let
-- Equations
  ec_0 = false -> pre s_is_open and close_;
  ec_1 = false -> pre (not s_is_open) and close_;
  ec_3 = false -> pre false and open_;
  s_is_open = true -> if ec_3 then pre s_is_open else (if ec_1 then pre true
  else (if ec_0 then pre false else pre s_is_open));
-- Constraints
  assert( close_ => (false -> pre (s_is_open or not s_is_open)) );
  assert( open_ => (false -> pre false) );
  assert( #(ec_0,ec_1,ec_3) );
  assert( not s_is_open or f_i = f_o );
  assert( #(open_,close_) );
tel
```

### 3.3.10 validate

Check a node against some basic validation properties.

#### Synopsis:

```
validate [--reach|--no-reach|id]*
```

Check each given node against some basic validation properties.

#### Description:

This command checks if a node verifies small validation properties. If such a property is not satisfied one can suspect an error or an incompleteness in the description of the node.

Some of these properties are checked on semantics of each given node and require the computation of reachable configurations. To disable the test of such properties one can specify the option `--no-reach`. Option `--no-reach` and `--reach` can be used several times and apply to following nodes.

Checked properties are the following:

#### Usage of variables

The command checks for each variable if it is used at least once i.e either in an assertion or in a transition.

#### Uniqueness of the initial configuration

The command verifies that only one initial configuration is possible. If this is not the case it reports variables that can take several values.

#### Coverage of domains w.r.t. configurations

The command checks that, according to assertions, each variable can be assigned all values in its domain. If this is not the case, the variable identifier and missing values are reported.

#### Coverage of domains w.r.t. reachable configurations

This is a similar same test than above but this time domains are checked against reachable configurations. If a variable does not cover its domain then missing values are reported but only values that are permitted by assertions.

#### Unused macro-transitions

The command reports macro-transitions (i.e built by flatten semantics) that are never triggered. This property requires reachable configurations.

*Example* : The following example displays the results of the command on a node whose variables does not cover their domain and a transition is enabled only from an unreachable configuration.

```
arc>eval
const N = 10;
const C_INIT = 1;

node A
  flow f : [0,2*N]
  state c : [0,N];
  init c := C_INIT;
  event inc, raz;
  trans
    c < N - 2 |- inc -> c := c + 1;
    c = N |- raz -> c := C_INIT;
  assert
    f = 2 * c;
edon
```



```

eval>EOF
arc>validate A
basic properties checking for node 'A'
there is 8 configurations.
usage of variables
  All variables are referenced at least once in assertions or transitions.

uniqueness of initial configuration
  The system has only one initial configuration

coverage of domains / configurations
  Flow variable 'f' does not cover its domain.
  Missing values (restricted to any assignments) verify:
    ((f = 5)) or ((f = 15)) or ((f = 17)) or ((f = 13)) or ((f = 1)) or ((f
    = 19)) or ((f = 11)) or ((f = 3)) or ((f = 7)) or ((f = 9))

coverage of domains / reachables
  State variable 'c' does not cover its domain.
  Missing values (restricted to configurations) verify:
    ((c = 0)) or (((9 = c)) or ((9 < c))) and (((c = 10)) or ((c < 10)))
  Flow variable 'f' does not cover its domain.
  Missing values (restricted to configurations) verify:
    ((f = 20)) or ((f = 0)) or ((f = 18))

usage of macro-transitions
  (c = 10) |- raz -> c := 1 is never triggered.

```

### 3.3.11 chkctl

Checks if initial configurations of a node satisfy a given CTL property.

#### Synopsis:

```
chkctl [--to-dot=filename] node F
```

Check if the CTL formula  $F$  is satisfied by initial configurations of *node*. Depending of the result, the command outputs a witness or a counter-example. If `--to-dot` specifies a valid filename, the result is displayed in DOT format.

## 3.4 Commands related to computations using exhaustive engine

### 3.4.1 ts

Display the transition systems of given nodes.

#### Synopsis:

```
ts id1 id2 ...
```

Displays transition systems of nodes  $id_1, id_2, \dots$

#### Description:

For each specified node, the command computes the transition system that represents its semantics. If the transition system has already been computed (with `ACHECK`, a call to `ts-marks`, ...) then it is not recomputed.

#### Remarks:

1. The transition system is displayed into the MEC 4 file format.
2. The transition system is represented in memory as an explicit graph and then displayed. All states and transitions are enumerated thus the output processes can take lots of time and place.

*Example* : The following example gives the transition system of a simple counter.

```

arc>eval
const MAX_VALUE = 3;
domain COUNTER_RANGE = [1, MAX_VALUE];

node Counter
  flow  out : COUNTER_RANGE;
  state value : COUNTER_RANGE;
  event inc, dec;
  trans true |- inc -> value := value + 1;
        true |- dec -> value := value - 1;
  assert out = value;
edon
arc>ts Counter
// transition system of node 'Counter'
transition_system Counter;
/*
 * # states = 3
 * # trans = 7
 */
out=1,value=1      |- '$'-> out=1,value=1;
                  |- inc-> out=2,value=2;
out=2,value=2      |- '$'-> out=2,value=2;
                  |- dec-> out=1,value=1;
                  |- inc-> out=3,value=3;
out=3,value=3      |- '$'-> out=3,value=3;
                  |- dec-> out=2,value=2;
<any_s = { out=1,value=1, out=2,value=2, out=3,value=3 }, empty_s = {
}, initial = { out=1,value=1, out=2,value=2, out=3,value=3 }>.

```

### 3.4.2 ts-marks

Lists properties computed with ACHECK engine.

#### Synopsis:

`ts-marks node-id`: Display properties of node *node-id*.

#### Description:

The command lists computed properties (i.e. sets of states and sets of transitions) for node identified by *node-id*; the cardinality of each set is displayed.

### 3.4.3 show-ts-marks

Display elements of properties computed using exhaustive engine on a transition system.

#### Synopsis:

`show-ts-marks nodeid  $P_1$   $P_2$  ...`

Display the elements of the sets  $P_1$ ,  $P_2$ , ... that are properties computed on the transition system associated with the node *nodeid*.

#### Description:

The command displays the content of sets (of states or transitions) computed on the transition system that represents the semantics of the given node.

If necessary this command computes the transition system for the given node.

## 3.5 Commands related to MEC 5 relations

### 3.5.1 card

Displays the cardinality of the relations given as arguments.

#### Synopsis:

`card  $id_1$   $id_2$  ...`

Display the cardinality of relations  $id_1, id_2, \dots$

**Description:**

This command displays the cardinality of the relations given as arguments.

*Example :*

```
arc>eval
eval>R(x : [0,10]) := <y : [0,10]> (x = 3 * y);
eval>EOF
R (x : [0, 10]) : 4 elements / 2 nodes
arc>card R
card (R) = 4
arc>
```

An argument can refer to a predefined relation that has not yet been computed.

*Example :*

```
arc>eval
const MAX_VALUE = 3;
domain COUNTER_RANGE = [1, MAX_VALUE];

node Counter
  flow out : COUNTER_RANGE;
  state value : COUNTER_RANGE;
  event inc, dec;
  trans true |- inc -> value := value + 1;
        true |- dec -> value := value - 1;
  assert out = value;
edon
arc>list relations
arc>card Counter!c
card (Counter!c) = 3
arc>list relations
defined relations : Counter!c
arc>
```

### 3.5.2 check-card

Check that the given properties have the expected cardinality.

**Synopsis:**

`check-card`  $id_1$  [ $card_1$ ]  $id_2$  [ $card_2$ ]  $\dots$

Checks that relation  $id_1$  has  $card_1$  elements,  $id_2$  has  $card_2$  elements,  $\dots$

**Description:**

This command checks that for each given relation  $id_i$  its cardinality is actually the one specified by  $card_i$ . If the cardinality is omitted the non-emptiness is checked. If the test fails, the actual cardinality is displayed between parenthesis.

Note that if the preference `arc.shell.check-card-abort` is true and if the cardinality of the set is not the expected one then the program is aborted. This feature is mainly used to make non-regression tests.

*Example :* The following example checks that a counter covers its domain. In a first time we simply check the property but then we show the abortion of ARC on an erroneous cardinality.

```
arc>eval
const MAX_VALUE = 3;
domain COUNTER_RANGE = [1, MAX_VALUE];

node Counter
  flow out : COUNTER_RANGE;
  state value : COUNTER_RANGE;
  event inc, dec;
```

```

    trans true |- inc -> value := value + 1;
           true |- dec -> value := value - 1;
    assert out = value;
edon
arc>check-card Counter!c 3
check card (Counter!c) = 3 passed
arc>check-card Counter!c 4
check card (Counter!c) = 4 failed (3)
arc>set arc.shell.check-card-abort true
arc>check-card Counter!c 4
check card (Counter!c) = 4 failed (3)
$ echo $?
1
$

```

See [\[set command\]](#), page 17 and preference [\[arc.shell.check-card-abort\]](#), page 71.

### 3.5.3 pick

Extract an element from an existing relation.

#### Synopsis:

```
pick rel_id [new_rel_id]
```

Display an element from the relation identified by *rel\_id* and put it in a new (singleton) relation *new\_rel\_id*.

#### Description:

This command creates a new relation *new\_rel\_id* with the same signature than *rel\_id* but containing just one element taken in *rel\_id* (if not empty). If the second argument is not given, the computed singleton is simply displayed and not stored.

```

arc>eval
domain Range = [-5, 5];

R(x : Range, y : Range) := x < 2 * y;
R (x : [-5, 5], y : [-5, 5]) : 58 elements / 8 nodes
arc>pick R Relement
x = -5, y = -2

Relement : [-5, 5] * [-5, 5] -> bool
1 elements/3 nodes
arc>show Relement
Relement contains :
x = -5, y = -2

arc>

```

### 3.5.4 store

Save a computed relation into a file.

#### Synopsis:

```
store "filename.rel" [rel-id1 [as new-id1]] [rel-id2 [as new-id2]] ...
```

Serializes relations *rel-id<sub>i</sub>* into the file called *filename.rel*.

#### Description:

This command permits to dump any relation into a file that can be reloaded later using the [\[load command\]](#), page 16. In order to reload relations, file names must be terminated with extension `.rel`. If the file exists then relations are stored at the end of the file.

For each *rel-id<sub>i</sub>*, dumped informations are the following:

- the signature of the relation;

- relations related to the signature (e.g. Main!c);
- the DD encoding the relation.

Each  $rel-id_i$  can be followed by a renaming using the syntax `as new-id`. In this case  $new-id$  is stored in place of  $rel-id_i$ .

*Example 1:* The following example creates a relation  $R$  and stores it into a file `R.rel`. The relation is then reloaded.

```
arc>eval
const MIN_VAL = 1;
const MAX_VAL = 6;
domain dom = [MIN_VAL, MAX_VAL];

R(s : dom) := <k : dom> (s = 3 * k);
eval>EOF
arc>show R
R contains :
s = 3
s = 6

arc>store "R.rel" R as Rprime
arc>list all
defined constants : MAX_VAL, MIN_VAL
defined domains : dom
defined relations : R
arc>load "R.rel"
arc>list all
defined constants : MAX_VAL, MIN_VAL
defined domains : dom
defined relations : R, Rprime
arc>info relations R Rprime
R : [1, 6] -> bool
cardinality          : 2
data structure size : 2

Rprime : [1, 6] -> bool
cardinality          : 2
data structure size : 2
arc>eval
eval>check_R() := [s : dom] (R(s) = Rprime(s));
eval>EOF
arc>show check_R
check_R contains :
true
arc>
```

The previous example works fine because the signature of  $R$  is not bound to any ALTARICA node. When a relation is related to a hierarchy of nodes, this hierarchy must be present when the relation is reloaded. The following example describes this point.

*Example 2:* First we store into `reach_Counter.rel` the relation encoding reachable configurations of a node `Counter`.

```
$ cat counter.alt
const MAX_VALUE = 3;
domain COUNTER_RANGE = [1, MAX_VALUE];

node Counter
  flow out : COUNTER_RANGE;
  state value : COUNTER_RANGE;
  event inc, dec;
  trans true |- inc -> value := value + 1;
        true |- dec -> value := value - 1;
  assert out = value;
```

```

edon
$ arc -q counter.alt
arc>store "reach_Counter.rel" Counter!reach
arc>info relations Counter!reach
Counter!reach : Counter!c -> bool
cardinality      : 3
data structure size : 1
arc>^D

```

Now we restart ARC with `reach_Counter.rel` as argument in order to reload the reachable configurations without any computation:

```

$ arc -q reach_Counter.rel
wrong data or missing info in file 'reach_Counter.rel'.

```

ARC displays an error because it misses types related to the node `Counter` e.g. `Counter!c`. To fix this lack we reload the description of the system (no computation is done for this step) and then the serialized relation:

```

arc>load "counter.alt"
arc>list all
defined constants : MAX_VALUE
defined domains  : COUNTER_RANGE
defined nodes   : Counter
arc>load "reach_Counter.rel"
arc>list relations
defined relations : Counter!c, Counter!reach
arc>show Counter!reach
out in [1, 3], value = out

arc>

```

## 3.6 Computation of sequences and fault trees: cuts and sequences

Computation of sets of scenarios

### Synopsis:

```

cuts [common options] nodeid observation
sequences [common options] [--order=k] nodeid observation

```

where common options are:

```

--visible-tags=id1,..., idn tags identifying observable events
--disabled-tags=id1,..., idn tags identifying not allowed events
--min compute minimal sequences
--enum enumerate sequences
--prefix=prefid prefix added to identifier of generated formulas

```

### Description:

These commands permit to compute sets of scenarios that lead the model described by *nodeid* into a configuration satisfying the formula *observation*. The formula *observation* can be any Boolean formula over all variables of *nodeid*.

Commands returns, by default, a Boolean formula that encodes implicants yielding the expected configurations. If `--enum` is specified, ARC enumerates elements of the result.

The formula is given using the syntax of the ARALIA tool. Literals of this formula are elementary events of the model that have been specified as *visible* by the user. *By default, the set of visible events is empty*; thus, the result of the command is 0 or 1 depending on the existence or not of expected configurations.

The following session shows an example of cuts computation. The studied node is a simple counter from 0 to 10. The counter can be incremented by one unit using `inc`

event or by two units using `inc2` event. The first event is labelled with attribute `attr1` and the second one with `attr2`. As explained above the result given by `cuts` is a Boolean constant (here 1 because expected states are reachable from initial state).

```
$ cat cuts-example.alt
node Counter
  state count : [0,10]; init count := 0;
  event inc : attr1;
      inc2 : attr2;
  trans true |- inc -> count := count + 1;
      true |- inc2 -> count := count + 2;
edon

$ arc -qb cuts-example.alt -c 'cuts Counter "count>=3"'
N0 := 1;
root := N0;
```

If we request ARC to compute scenarios but considering that `inc` and `inc2` must appear in the result we obtain a Boolean formula that encodes two sequences:

```
$ arc -qb cuts-example.alt -c 'cuts --visible-tags=attr1,attr2 Counter "count>=3"'
N2 := 1;
N1 := ('inc2' ? -N2 : N2);
N0 := ('inc' ? -N2 : N1);
root := -N0;
```

Now if only event `inc2` is observed we obtain yet the Boolean constant 1.

```
$ arc -qb cuts-example.alt -c 'cuts --visible-tags=attr2 Counter "count>=3"'
N0 := 1;
root := N0;
```

Why do we obtain 1? Because, event `inc2` is implicitly simplified. Actually, non-observed events are projected (i.e., quantified) in the formula obtained when all events are observed. In our example the formula is `inc` or `inc2`; thus when `inc2` is quantified the formula is simplified into 1.

The option `--disabled-tags=id1,...` permits to indicate that events labelled with tags `idis` are forbidden in scenarios:

```
$ arc -qb cuts-example.alt -c 'cuts --visible-tags=attr2 --disabled-tags=attr1
Counter "count>=3"'
N1 := 1;
N0 := ('inc2' ? -N1 : N1);
root := -N0;
```

The command `cuts` computes sets of events, called *cuts*, which means that the order of occurrence of events does not appear in the result. If this order is relevant, one can use the command `sequences` that receives the same arguments than `cuts` except an additional one the maximal length of computed sequences. The option `--order=k` indicates that ARC has to compute ordered sequences of events that can not contain more than *k* visible events. If we come back on previous example and limit the number of visible events to 3 we get sequences given below. Note that this time ordered sequences are not given as Boolean formulas but enumerated (we used `--enum`).

```
$ arc -qb cuts-example.alt -c 'sequences --enum --visible-tags=attr1,attr2
--ordered=3 Counter "count>=3"'
(inc2, inc2)
(inc2, inc)
(inc, inc2)
(inc, inc, inc2)
(inc, inc, inc)
```

One can notice that some sequences are missing; actually, we could complete those of length 2 with a third event and always get `count>=3`. Indeed, the algorithm translates the decision diagram of reachable configurations into sequences and some redundant

occurrences of events may have been suppressed by DD construction rules. These missing sequences are not actually interesting because they contain shorter sub-sequences that produce the expected configurations.

The option `--min` can be used to filter sets to minimal elements. If cuts are computed the minimality criterion is the inclusion. In the case of ordered sequences, sub-sequences (or sub-words) are considered.

```
$ arc -qb cuts-example.alt -c 'sequences --enum --visible-tags=attr1,attr2
--ordered=3 --min Counter "count>=3"'
(inc2, inc2)
(inc2, inc)
(inc, inc2)
(inc, inc, inc)
```

### 3.7 Stochastic simulation: `sas`

The stochastic simulator is invoked using the `sas` command. For detail on stochastic simulation in ARC we refer the user to [Chapter 6 \[Stochastic simulation\], page 55](#).

#### Synopsis:

```
sas [options]
```

where *options* are:

```
--scheduler=S
```

specifies the data-structure used to implement the scheduler of events. *S* can take two values:

- `cq` for a calendar queue implementation ([\[RB98\], page 69](#)).
- `dlink` for a doubled-linked list scheduler.

By default `dlink` is used.

```
--prng=R selects the pseudo-random number generator. R can take 3 values:
```

- `ed` for the Erard-Deguenon generator
- `mks` is the Kiss-SWB generator of Marsaglia.
- `mk1` is the Kiss-LFIB4 generator of Marsaglia.

By default `ed` is used.

```
--nb-threads=P
```

*P* indicates the number of threads used to simulate the system. Each thread executes  $N/P$  stories where *N* is the total number of stories.

```
--ignore-sigint
```

This option allows the cancellation of ARC when the user sends a INT signal to the process (e.g. using `Ctrl-C`). When this option is omitted, ARC displays a menu that proposes the user either to cancel the simulation or to display current results or to continue the simulation.

```
--timeout=d
```

*d* permits to allocate *d* seconds to the simulation. After this time ARC terminates the simulation of current stories and then cancel the whole simulation.

```
--seed=s s specifies an integer seed s for the pseudo-random number generator.
```

```
--loop-length=len
```

*len* indicates the length of the longest sequence of transitions that does not change the current time. If the simulator executes *len* transitions without increasing the time, ARC considers it enters an infinite loop and stop the simulation.



`--nb-stories=N`

$N$  is the number of stories that have to be simulated. Each story is an execution of the system.

`--duration=T`

$T$  is the number of time units for each story.  $T$  depends on the nature of the system and parameters of laws. For instance if failure rates are given in hours, the time allowed for the mission of the system  $T$  should be expressed in hours.

`--nb-tries=t`

If parameters of laws are not constants, ARC computes  $t$  simulation with  $N$  stories. Before each simulation of  $N$  stories, the tool draws new values for random parameters.

### Description:

`sas` command is a Monte-Carlo algorithm that simulates random behaviors of a system according to stochastic data given in the `extern` clauses of its AltaRica model. For a detailed description of these clauses, we refer the user to [Chapter 6 \[Stochastic simulation\], page 55](#).

The algorithm simulates  $N$  runs (specified with `--nb-stories`) from the initial configuration of the system until the time reach the limit  $T$  specified with the option `-duration`. Each run is called a *story*.

For each story, ARC records statistical data related to the execution of the system among which the frequency of transitions. The user can also specify observers. An observer is either a Boolean formula (`predicate`) over variables of the system or an simple expression (`property`) e.g the value of an integer variable. For each Boolean observer ARC computes the mean, the standard deviation of the following random variables:

- The number of times (per story) the observer is equal to `true`;
- The cumulated time with the value `true`;
- The first instant where the observer is equal to `true`.
- The number of missed occurrences of the observer i.e. when the observer is true after the mission delay.

Else, for all observers, the tools computes:

- Its value;
- Its final value.

For each measure, the tool reports its *confidence* to 90%. If the mean is  $m$  while the actual mean is  $M$ , ARC computes  $\epsilon$  such that  $Prob(M - \epsilon \leq m \leq M + \epsilon) = 90\%$ .

*Example* : The following example is a simple component that can fail with a rate of one failure for 1000 hours. It can also be repaired with a rate of one per 100 hours.

```
node ComponentLambdaMu
  event
    failure, repair;
  state
    mode : { OK, KO }; init mode := OK;
  trans
    mode = OK |- failure -> mode := KO;
    mode = KO |- repair -> mode := OK;
  extern
    parameter LAMBDA = 1e-3;
    parameter MU = 1e-2;

    law <event failure> = exponential(LAMBDA);
```

```

    law <event repair> = exponential(MU);
    predicate KO = <term (mode = KO)>;
    edon

```

```

sas --duration=10000 --nb-stories=1000000 --nb-threads=4 ComponentLambdaMu

```

We simulated the behaviors of this component and observed its failure mode. ARC gives us the following results:

```

*** ACTION FREQUENCIES
      NAME           MEAN           STDDEV           CONF.
  failure      9.09434E+00      2.75802E+00      4.52315E-03
  repair       9.00353E+00      2.74571E+00      4.50296E-03

*** CUMULATED TIME WITH EXPECTED VALUE
      NAME           MEAN           STDDEV           CONF.
      KO            9.00726E+02      3.84160E+02      6.30023E-01

*** NUMBER OF OCCURRENCES
      NAME           MEAN           STDDEV           CONF.
      KO            9.09434E+00      2.75802E+00      4.52315E-03

*** FIRST OCCURRENCES
      NAME           MEAN           STDDEV           CONF.
CENSURED
      KO            1.00060E+03      9.99918E+02      1.63992E+00
61

```

## 3.8 Experimental commands

### 3.8.1 diag

Computation of fault trees.

#### Synopsis:

```

diag [--visible-tags=id1,...] [--disabled-tags=id1,...] nodeid observation

```

#### Description:

Similarly to [\[cuts command\]](#), [page 33](#), this command aims to produce fault-trees of an ALTARICA model wrt a given unexpected configuration *observation*. Introduced in ARC 1.5, this new command uses an **algorithm that is not yet guaranteed** and thus it must be used in full knowledge of that fact.

While *cuts* is a global algorithm that computes reachable configurations of the model, *diag* has an approach based on hand-made fault trees which basically is not able to capture dynamic behaviors.

Arguments are the same that those of [\[cuts command\]](#), [page 33](#) except that *--enum* and *--min* are not allowed.

*diag* produces a Boolean formula as a set of equations given in ARALIA format.

### 3.8.2 sat

Use internal SAT solver to check satisfiability of formulas.

#### Synopsis:

```

sat node cnf F
sat node solve F
sat node assertions
sat node steps k
sat node reachables F k

```

**Description:**

This command is used to make experiments with the internal SAT Solver which is currently GLUCOSE (see [al15], page 69). The first argument of the command is an ALTARICA node on which the command is applied. Then following sub-commands can be specified (a prefix of it can be used):

`cnf  $F$`  The formula  $F$  is compiled into a SAT instance in conjunctive normal form. This command display the Boolean formula and its corresponding set of clauses sent to SAT solver.

`solve  $F$`  The SAT solver looks for a model (a solution) of the Boolean formula  $F$ . This latter is built over variable of the input *node*. The command display `sat` if a model has been found or `unsat` is the formula is not satisfiable.

`assertions` Apply the satisfiability checking on assertions of the node. The command outputs `sat` or `unsat`.

`steps  $k$`  This command produces a Boolean formula that represents a sequence of  $k$  steps from an initial configuration and checks if the formula is satisfiable. The underlying sequence of states does not contain a loop (i.e. all states are different).

`reachables  $F$   $k$`  This command checks if there exists a configuration that satisfies the formula  $F$  in less than  $k$  steps.

## 4 Using the ACHECK specifications

In this chapter we present the ACHECK module of ARC. This part of the tool is inherited from ACHECK, the model-checker of the previous suite of tools called ALTATools. The main difference with the ACHECK tool is that ARC use two data structures to represent state-spaces; either the explicit state graph defined by the semantics of nodes, or relations stored as BDDs that represent sets of configurations or sets of transitions.

### 4.1 Overview

We have kept the input language of ACHECK for specifications. An ACHECK file is a list of blocks

```
with N1
, ..., Nk
do
...
done
```

where  $N_i$  are identifiers of nodes loaded into memory. A list of computation queries or commands are placed between the `do` and `done` keywords. All commands or computations are all applied to a node  $N_i$  and nodes are treated sequentially. Each command or computation implicitly refers to the node under study. For instance, the following command:

```
dot (any_s, any_t)
```

displays the state-graph of the current node using the `dot` graph file format.

Comments can be added using C-like comments:

```
with N do /* a C comment can be used for one */
...
/* or for
several
lines */
...
done
```

or C++ like comments:

```
with N do // a C++ comment starts after the double-slash and
... // ends at the end of the line.
done
```

While computations do not produce any output, commands (e.g `dot`) print information onto the standard output of the ARC process. Data displayed by commands can be redirected. This redirection uses the same notation that the one used by Unix shells:

```
> 'filename'
```

redirects the stream into the file *filename*. The file is created if necessary; but if the file already exists then its content is erased.

```
>> 'filename'
```

redirects the stream at the end of the file *filename*. The file is created if necessary.

If `$NODENAME` is a substring of *filename* then this latter is replaced by the name of the node under study. For instance the following ACHECK command:

```
with Switch, Generator do
show (all) > '$NODENAME.result';
done
```

should create two files named `Switch.result` and `Generator.result` that contain computed predefined properties for nodes `Switch` and `Generator`.

## 4.2 Representation of the semantics

As said in the introduction of this chapter, ACHECK supports two kinds of data structures to store state-spaces. The very first version of the ACHECK engine uses an explicit representation of state-spaces using an explicit graph as data structure. The Decision Diagrams (DD) package of the TOUPIE tool ([cr97], page 69) has been then integrated into ARC. While edges of a BDD are labelled by Boolean values 0 and 1, those of a DD are labelled by the  $n$  values in the domain of the considered variable. Thus, DDs are essentially a  $n$ -ary extension of BDDs.

The default data structure used for computation of properties is the DD representation. It is possible to explicitly specify the data structure using a keyword after `do`: `symbolically` for DDs or `exhaustively` for graphs.

```
with N1
, ..., Nk
do exhaustively // enable graph-based representation
...
done
```

Depending on the chosen data structure used to store state spaces, some commands, built-in properties or operators are not available. In the sequel we indicate this availability using: *TS* to denotes the graph data structure and *DD* for Decision Diagrams.

## 4.3 Computing properties of nodes

ACHECK language allows to specify sets of configurations and sets of transitions. Each set is defined using an equation of the form:

$$X @ = F$$

where  $X$  is the name assigned to the computed set and  $F$  is a formula defining the set and which depends on already computed sets. @ must be replaced by a colon (:), a plus (+) or a minus (-) symbol:

- $X := F$  defines a set that is computable directly from the evaluation of the formula  $F$ .
- $X += F(X)$  or  $X -= F(X)$  define the set  $X$  as, respectively, the least and the greatest fixed point<sup>1</sup> of the monotone function  $F(X)$ . Of course, if  $X$  does not appear in the formula  $F$  then  $X := F$ ,  $X += F$  and  $X -= F$  define the same set  $X$ .

Of course, depending on the underlying data structure, algorithms used to compute sets are different. On the one hand, if the explicit state-graph is used, the linear algorithm of Arnold and Crubillé is called to solve fixed point equations ([ac88], page 69). On the other hand, when Decision Diagrams ([cr97], page 69) are used, the function defining the set is simply iterated until the fixed point is reached.

Formulas describe either sets of transitions or sets of configurations. Formulas are built using predefined or user defined sets, Boolean operators and built-in function that compute either transitions or configurations sets. All these elements are further detailed in sub-sections below.

**Remark:** The reader should keep in mind an important difference between the two representations: when the explicit representation is used the computed sets are sets of **reachable** objects (states/configurations or transitions) while this is not the case for the sets encoded with DDs.

### 4.3.1 Built-in sets

ACHECK pre-defines several sets of configurations or transitions. These sets are given below. For each one we indicate its availability w.r.t. the selected encoding of the semantics.

<sup>1</sup> The += and -= notations (inherited from TOUPIE) refers to the way the computed set evolves at each iteration of the fixed point computation (when using such algorithm). Least fixed point add (+) new elements to the set while the greatest fixed point removes (-) elements.

### 4.3.1.1 Sets of configurations

**any\_c** This set contains all valid configurations of the current node. A configuration is an assignment of variables that satisfies the global assertion of a node. Remember that a configuration is not necessarily reachable from the initial state(s).

**Encoding:** *DD*

**valid\_state\_assignments**

This set contains assignments of variables such that the value of state variables permits to satisfy assertions of the node. Flow variables are allowed to take any value in their domains (thus, this set is **not** a subset of valid configurations **any\_c**).

**Encoding:** *DD*

**empty\_s** This set contains no configuration.

**Encoding:** *TS, DD*

**any\_s** This set contains configurations that are reachable from initial configurations.

**Encoding:** *TS, DD*

**initial** This set contains initial configurations i.e. assignments of variables that satisfy both the **init** and **assert** clauses of the node. By definition this set is a subset of **any\_c**.

**Encoding:** *TS, DD*

### 4.3.1.2 Sets of transitions

**any\_t** This set contains transitions between reachable configurations i.e. elements of **any\_s**.

**Encoding:** *TS, DD*

**any\_trans**

This set is the relation transition where post- and pre- conditions are not applied.

**Encoding:** *DD*

**empty\_t** This set contains no transition.

**Encoding:** *TS, DD*

**epsilon** This set of transitions is the restriction of **any\_t** to transitions labelled by the  $\epsilon$  event or, more precisely, transitions where all components trigger the  $\epsilon$  event.

**Encoding:** *TS, DD*

**not\_deterministic**

This set identifies not deterministic transitions. A transition  $(s, e, s')$  is not deterministic if there exists another transition  $(s, e, s'')$  such that  $s' \neq s''$ .

**Encoding:** *TS*

**self** This is the set of elementary loops i.e. transitions with the same (reachable) configuration as source and target.

**Encoding:** *TS, DD*

**self\_epsilon**

This set is simply a shortcut for  $self \cap epsilon$ .

**Encoding:** *TS, DD*

**valid\_state\_changes**

This is the restriction of **any\_trans** to **valid\_state\_assignments**.

**Encoding:** *DD*

### 4.3.2 Operators

`assert( $S$ )`

$S$  denotes a set of assignments of variables that are not necessarily configurations. `assert` intersects  $S$  with assertions of the model (i.e. `any_c`) which makes the result valid configurations.

If the encoding is *DD* the result is not necessarily a subset of `any_s` because computed configurations can be unreachables.

**Encoding:** *TS, DD*

$X_1$  and  $X_2$

$X_1 \& X_2$  Computes the intersection of the two sets  $X_1$  and  $X_2$ . The two sets must have the same type.

**Encoding:** *TS, DD*

$X_1$  or  $X_2$

$X_1 \mid X_2$  Computes the union of the two sets  $X_1$  and  $X_2$ . The two sets must have the same type.

**Encoding:** *TS, DD*

$X_1 - X_2$

Computes the difference of the two sets  $X_1$  and  $X_2$ . The two sets must have the same type.

**Encoding:** *TS, DD*

$[\phi]$

This operator returns assignments of variables that satisfy  $\phi$  where  $\phi$  is a Boolean AltaRica expression over variables of the considered node.

Note that depending on the encoding of the state space,  $[\phi]$  has not the same semantics. On the one hand, when the encoding is *TS* the result contains assignments that are, by construction, reachable configurations. On the other hand, the result computed using the *DD* encoding is not constrained to belong to `any_s` or `any_c`.

**Encoding:** *TS, DD*

`coreach( $S, T$ )`

$S$  denotes a set of configurations and  $T$  a set of transitions. `coreach` computes the set of valid configurations (i.e. belonging to `any_c`) that are co-reachable from  $S$  using only transitions in  $T$ .

If the encoding is *DD* the result is not necessarily a subset of `any_s`.

**Encoding:** *TS, DD*

`label  $E$`

$E$  is the identifier of an elementary event. `label  $E$`  returns the set of transitions whose (global) events contain  $E$ . If the encoding is *DD*, transitions are not constrained by pre- or post-conditions i.e. `label  $E$`  is a subset of `any_trans`.

**Encoding:** *TS, DD*

`attribute  $A$`

$A$  is the identifier of an attribute that labels events. `attribute  $A$`  returns the set of transitions whose (global) events possess at least one event labelled with  $A$ . If the encoding is *DD*, transitions are not constrained by pre- or post-conditions i.e. `attribute  $A$`  is a subset of `any_trans`.

**Encoding:** *TS, DD*

`loop( $T_1, T_2$ )`

$T_1$  and  $T_2$  are sets of transitions. The `loop` operator returns the subset of  $T_2$  that form strongly connected components (not necessarily one) containing at least one transition in  $T_1$ .

- Encoding:** *TS*
- `not X` *X* is either a set of configurations or a set of transitions. This operator returns the complement of the given set *X*. If the encoding is *TS*, *X* is complemented into reachable configurations (i.e. `any_s`) or transitions (i.e. `any_t`). If the encoding is *DD* the complement is taken from the set of all possible assignments or transitions.
- Encoding:** *TS, DD*
- `pick(X)` This operator simply returns a singleton taken from *X* or an empty set if *X* is empty. The way the element is selected is not specified.
- Encoding:** *TS, DD*
- `proj_f(X)`  
`proj_s(X)`
- These operators project the given relation *X* on a subset of variables: flow variables for `proj_f` and state variables for `proj_s`. Resulting relations returned by `proj_x` have the same arity than *X* i.e. variables are not actually removed but they are allowed to take any value in their domain.
- X* can be either a set of states or a set of transitions.
- Encoding:** *DD*
- `reach(S, T)`
- This operator returns the set of valid configurations that are accessible from *S* using transitions in *T*. Note that if the encoding is *DD* the computed configurations are not necessarily reachable from initial ones.
- Encoding:** *TS, DD*
- `rsrc(S)` This operator returns the set of transitions enabled from a configuration in *S*. In the case of a *DD* encoding, transitions are not constrained by pre- and post-conditions.
- Encoding:** *TS, DD*
- `rtgt(S)` This operator returns the set of transitions that lead to a configuration in *S*. In the case of a *DD* encoding, transitions are not constrained by pre- and post-conditions.
- Encoding:** *TS, DD*
- `src(T)` This operator returns the set of configurations that are the origin of at least one transition in *T*. Returned configurations are not constrained by assertions.
- Encoding:** *TS, DD*
- `tgt(T)` This operator returns the set of configurations that are the target of at least one transition in *T*. Returned configurations are not constrained by assertions.
- Encoding:** *TS, DD*
- `trace(S1, T, S2)`
- This operator returns a shortest path from configurations in *S<sub>1</sub>* to those belonging to *S<sub>2</sub>* and using transitions in *T*. The result of `trace` is the set of transitions that compose the path.
- In the case of *DD* encoding the path is not necessarily reachable from the initial state.
- Encoding:** *TS, DD*
- `unav (T, S)`
- This operator returns the set of configurations from which all paths composed of transitions in *T* pass by a configuration in *S*.
- Encoding:** *TS, DD*



### 4.3.3 Using CTL\* logic

When using DD encoding, ACHECK permits to specify sets of states using the well-known logic CTL\* ([eh86], page 69). As usual, formulae are built from previously computed sets and composed with Boolean connectives (&/and, |/or, ~/not, =>, <=> and xor). The CTL\* logic allows to talk about paths starting from states. A quantifier of path E is introduced: if  $\phi$  is a path-formula (see below), a state  $s$  satisfies the formula  $E[\phi]$  if *there exists* a path starting from  $s$  that satisfies  $\phi$ .

Formulae about paths are built using state formulae, Boolean connectives, an unary operator X (called *next*) and a binary one U (called *until*). Operands of X and U are either state or path formulae. If  $p$  is a state formula,  $\psi$  and  $\phi$  path formulae, then a path  $\sigma = s_0, s_1, s_2, \dots$  satisfies:

- $p$  if  $s_0$  satisfies  $p$ ;
- $X\psi$  if the suffix  $s_1, s_2, \dots$  satisfies  $\psi$  which means intuitively that at the next step the path satisfies  $\psi$ ;
- $\psi U \phi$  if there exists  $i \geq 0$  such that for all  $0 \leq j < i$ ,  $s_j, \dots, s_i, \dots$  satisfies  $\psi$  and  $s_i, \dots$  satisfies  $\phi$ . Intuitively the formula expresses that the path satisfies  $\psi$  *until*  $\phi$  becomes true.

From basic operators several shortcuts are defined:

- $A[\psi] \equiv \text{not } E[\text{not } \psi]$  is satisfied by states from which all paths satisfy  $\psi$ .
- $F\psi \equiv \text{true } U \psi$  is satisfied by paths that have a suffix that satisfy  $\psi$ . F abbreviates *finally* and means that something eventually happens in the future on the path.
- $G\psi \equiv \text{not } F(\text{not } \psi)$  is satisfied by paths whose all suffixes satisfy  $\psi$ . G abbreviates *globally* and means that something is always true all along the path.
- $\psi W \phi \equiv (\psi U \phi) \text{ or } G\psi$  is called the *weak until*. It specifies that  $\psi$  is true until  $\phi$  becomes true but, if  $\phi$  can not be satisfied then  $\psi$  must be always true.
- $AX\psi \equiv A [ X \psi ]$  is satisfied by states such that at the next instant  $\psi$  is satisfied.
- $AF\psi \equiv A [ F \psi ]$  is satisfied by states from which eventually in the future  $\psi$  is satisfied.
- $AG\psi \equiv A [ G \psi ]$  is satisfied by states from which  $\psi$  is always satisfied. If an initial state satisfies  $AG\psi$  then  $\psi$  is an *invariant* of the system.
- $EX\psi \equiv E [ X \psi ]$  is satisfied by states from which at the next step  $\psi$  can be satisfied.
- $EF\psi \equiv E [ F \psi ]$  is satisfied by states from which  $\psi$  can be satisfied in the future.
- $EG\psi \equiv E [ G \psi ]$  is satisfied by states from which  $\psi$  can become continuously true.

In order to specify a set using CTL\* logic, the formula must be preceded by the `ctlspec` keyword. Note that CTL\* specifications are not allowed within fixed-point equations.

```
with N do
  P := ...;
  C := ctlspec E [ X P ];
done
```

No dedicated decision procedure is implemented to check CTL\* formulae. Actually each CTL\* formula is translated into a system of fixed-point equations that is solved using Mec 5 engine (see Chapter 5 [Using the Mec 5 specifications], page 49). The translation procedure is those given in [mm94], page 69. The equation system used to check a CTL\* formula can be displayed using ACHECK command [ct12mu], page 45.

```
arc>eval
eval>node A edon
with A do
  ct12mu (AX any_s);
done
eval>EOF
translation of AX any_s into Mec 5 specs:
begin
  local X_0 (v~1 : A!c) += A!any_s (v~1) & <t~4 : A!c>A!nsemove (v~1, t~4) & ~A!any_s (t~4);
```

```

R$ (v~5 : A!c) +2= A!any_s (v~5) & ~X_0 (v~5);
end
arc>

```

In the previous example `R$` is an internal identifier used to name the relation encoding the computed set.

## 4.4 Commands

`display(id1, ..., idn)`

This command lists all the elements of sets  $id_1, \dots, id_n$ . Each  $id_i$  is either a predefined set like `any_s` or a set computed by the user. If the encoding is `DD` the predefined sets are computed on demand.

Since the command enumerates elements of specified sets, it is quite wasteful to use it on large sets. To get the size of the set, the command `show` (see [\[show\]](#), page 46) should be preferred and the operator `pick` (see [\[pick\]](#), page 43) can be used to extract samples from sets.

**Encoding:** `TS, DD`

`ctl2mu(F)`

This command outputs the translation of CTL\* formula  $F$  into a MEC 5 specification. See [Section 4.3.3 \[Using CTL\\* logic\]](#), page 44.

**Encoding:** `DD`

`dot(S, T)` This command outputs, in DOT graph format, the reachability graph restricted to the set of configurations  $S$  and transitions  $T$ . Note that all reachable configurations belonging to  $S$  are displayed even if they have no successors or predecessors.

**Encoding:** `TS, DD`

`dot-trace(I, T, F)`

This command outputs, in DOT graph format, a shortest trace from a state in  $I$  to a state in  $F$  using transitions in  $T$ . Actually this command is just a shortcut for:

```

tr := trace (I, T, F);
dot (src(tr) or tgt(tr),tr);

```

**Encoding:** `TS, DD`

`events(T)`

This command lists events that labels transitions belonging to the set  $T$ .

**Encoding:** `TS, DD`

`modes()`

This command display the mode-automaton of the current node. A *mode* is an assignment of state variables. Configurations are gathered according to each mode.  $T$ .

**Encoding:** `DD`

`nrtest('filename')`

`nrtest(X, n)`

This command is used to realize non-regression tests. In its first form, the user specifies in a file *filename* expected results. Each line of this file is a couple:

```

set-identifier cardinality

```

The `nrtest` command simply checks the equality of the cardinality of the computed set `set-identifier` with the one specified into *filename*.

If either the set has not been computed or if the cardinalities differ the test fail and in this case, according to the value of preference `acheck.nrtest-failure-aborts`, the program might terminate and returns to the caller program (e.g. the shell) with an error code.

The second form simply checks that  $n$  is the cardinality of set  $X$ .

See preference [`acheck.nrtest-failure-aborts`], page 72.

**Encoding:** *TS, DD*

```
project(S, (TE, TA), id, simplify [, subnode])
project(S, TE, id, simplify [, subnode])
```

This command projects the semantics of the node under study on one of its component if the argument *subnode* is specified; else it is projected on itself. The displayed result is an ALTARICA node named *id* that is essentially the same than the original one but where transitions are restricted with those actually reached when the semantics is computed. The reachability graph can also be restricted to configurations belonging to  $S$  and transitions of  $TE$  and  $TA$ . If the Boolean argument *simplify* is `true` then guards are simplified; the aim of this parameter is to obtain clearest transitions (in particular for trivial guards).

**Encoding:** *TS, DD*

```
quot()
```

This command displays, in `dot` graph format, the greatest autobisimulation compatible with already computed properties of configurations; in other words the greatest bisimulation included in the relation

$$R = \{(s, s') | \forall P \in \mathcal{P}, s \in P \iff s' \in P\}$$

where  $\mathcal{P}$  is the set of configuration sets already computed.

**Encoding:** *TS, DD*

```
remove(id1, ..., idn)
```

This command removes from ARC memory properties specified by identifiers  $id_1, \dots, id_n$ . If some property  $id_i$  does not exists a warning is emitted.

**Encoding:** *TS, DD*

```
show(id1, ..., idn)
```

This command displays the cardinality of sets identified by  $id_i$ s. If some  $id_i$  is equal to `all` then the command displays all already computed sets.

In the case of *DD* encoding, predefined sets (e.g. `any_s`) are computed on demand. Note that the `all` keyword does not induce the computation of predefined sets.

**Encoding:** *TS, DD*

```
test(X, n, ce)
test(X, n)
test(X, w)
test(X)
```

This command is used to check that the cardinality of the set  $X$  is  $n$ . The command simply displays the result of the test. If the test fails the actual size is displayed. If the set is expected to be empty (i.e.  $n$  should be 0) and if the Boolean *ce* is `true` while the test *fails*, a counter-example is computed and displayed.

If the expected cardinality  $n$  is not specified the non-emptiness of  $X$  is checked and if *w* is `true` while the test *successes*, a witness is picked from the set and displayed.

**Encoding:** *TS, DD*

`validate()`

This command simply applies [`validate command`], page 27 to the current node.

**Encoding:** *DD*

`wts(S, T)`

This command is inherited from MEC 4. It displays the restriction of the transition system in MEC 4-like format.

**Encoding:** *TS*



## 5 Using the MEC 5 specifications

MEC 5 (see [av03], page 69) files consist in the definition of predicates (also called *relations* in the sequel). Files may contain domain or constant definitions using the ALTARICA syntax.

This chapter is divided into two sections. The first one describes the syntax used to define new relations. The last one presents a set of predefined relations that are bound to each ALTARICA node.

### 5.1 Writing MEC 5 predicates

The definition of a  $n$ -ary predicate called *name* has the following form:

$$\text{name}(p_1 : \text{dom}_1, \dots, p_n : \text{dom}_n) \xi = \phi(p_1, \dots, p_n);$$

where:

- $p_i$ s are formal parameters of the predicate. Each  $p_i$  takes its values into the domain  $\text{dom}_i$ .
- $\phi$  is a Boolean formula that depends on  $p_i$ s and on already defined predicates and constants. The syntax of Boolean formula is the same than for ALTARICA except that quantifiers are allowed. The following formula:

$$\langle x_{1,1}, \dots, x_{1,n_1} : D_1; \dots; x_{k,1}, \dots, x_{k,n_k} : D_k \rangle F$$

means

$$\exists x_{1,1} \in D_1, \dots, \exists x_{1,n_1} \in D_1, \dots, \exists x_{k,1} \in D_k, \dots, \exists x_{k,n_k} \in D_k (F).$$

Square brackets, [ and ], are used in place of < and >, for the universal quantification ( $\forall$ ).

- $\xi$  is one of the following characters:
  - a colon (:) which means that the defined *name* is simply the set of assignments of  $p_i$ s that satisfy  $\phi$ .
  - a plus sign (+) which means that *name* is defined recursively in function of itself and its semantics is the least fixed point of the recursive sequence.
  - a minus sign (-) which means that *name* is defined recursively in function of itself and its semantics is the greatest fixed point of the recursive sequence.

*Example 1:* The following example gives two ways to define a predicate that is true only for even numbers between 0 and 20. The first relation specifies that an even number is a multiple of 2 while the second one specifies that an even number is simply equal to an even number plus 2. We check that both relations are equivalent using a predicate named *diff* containing integers that differentiate both versions of the *even* predicate.

```
arc>eval
eval>const N = 10;
domain R = [0, N];
even_v1(s : [0, N]) := <n : [0,N]> (s = 2*n);
even_v2(s : [0, N]) += (s=0) | <t : [0,N]> (even_v2 (t) and s = t+2);
diff(s : [0,N]) := not (even_v1(s) = even_v2(s));
eval>EOF
even_v1 (s : [0, 10]) : 6 elements / 2 nodes
even_v2 (s : [0, 10]) : 6 elements / 2 nodes
diff (s : [0, 10]) : empty
arc>show even_v1
even_v1 contains :
s = 0
s = 2
s = 4
s = 6
s = 8
s = 10

arc>
```

Sometimes it is convenient to split up an equation into several clearest equations. Equations systems are specified between **begin** and **end** keywords.

*Example 2:* Similarly to example 1, the following equation system defines two predicates that are true, respectively, when an integer in some range  $R$  is *odd* or *even*. The definitions of these predicates are mutually recursive: an odd number is equal to an even number plus one and an even number is either 0 or is equal to an odd number plus one.

```
arc>eval
eval>const N = 10;
domain R = [0, N];

begin
  odd (s : R) += <t : R> (even(t) and s = t + 1);
  even (s : R) += (s = 0) | <t : R> (odd (t) and s = t + 1);
end
eval>EOF
even (s : [0, 10]) : 6 elements / 2 nodes
odd (s : [0, 10]) : 5 elements / 2 nodes
arc>show even
s = 0
s = 2
s = 4
s = 6
s = 8
s = 10

arc>show odd
s = 1
s = 3
s = 5
s = 7
s = 9

arc>
```

Some equations defined in equation systems exist only for the sake of clarity. It is possible to introduce temporary definitions whose names and allocated memory resources are released after the computation of the fixed point. Temporary definitions are introduced using the **local** keyword.

*Example 3:* The previous predicates, *even* and *odd*, can be defined using a third binary predicate *succ*( $s, t$ ) that is true when  $t = s + 1$ . If we consider that *succ* is not important for the sequel it can be set local to the equation system:

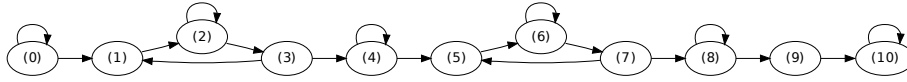
```
arc>list all
arc>eval
const N = 10;
domain R = [0, N];

begin
  local succ (current : R, next : R) := next = current + 1;
  odd (s : R) += <t : R> (even(t) and succ (t, s));
  even (s : R) += (s = 0) | <t : R> (odd (t) and succ (t, s));
end
eval>EOF
even (s : [0, 10]) : 6 elements / 2 nodes
odd (s : [0, 10]) : 5 elements / 2 nodes
arc>list all
defined constants : N
defined domains : R
defined relations : even, odd
arc>
```

MEC 5 equation systems also permit to alternate types of fixed point. Alternation is introduced using a positive integer  $d$  between the sign and the equal characters:  $+d=$  or  $-d=$ . This

integer indicates the height of the  $\mu/\nu$  operator into the alternation hierarchy; the higher  $d$  is, the higher the operator is.

*Example 4:* The following example is taken from TOUPIE and MEC 5 manuals. Below is a graph whose vertices are labelled with integers.



One wants to compute states from which there exists an infinite path containing a vertex with an odd label. States verifying this property satisfy the formula  $\nu T. \mu A (\diamond A \vee \diamond (T \wedge ODD))$ .

```
arc>eval
eval>const N = 10;

domain vertex = [0,N];

g(S : vertex, T : vertex) := (T = S + 1)
                             | ((S = 3 | S = 7) & (T = S - 2))
                             | ((<n : [0, N]> (S = 2 * n)) & (T = S))
                             ;
odd (S : vertex) := <n : [0, N]> (S = 2 * n + 1);
begin
  local aux(V : vertex) +1= <W : vertex>(g(V, W) & aux(W))
                             | <W : vertex>(g(V, W) & odd(W) & tau(W));
  tau(U : vertex) -2= aux(U);
end
eval>EOF
arc>show tau
tau contains :
U in [0, 7]
```

## 5.2 Built-in MEC 5 relations

MEC 5 allows the use of particular identifiers that use an exclamation mark (!). This special identifiers represent either a relation (e.g. !init) or a type (e.g. !c) bound to an ALTARICA node. Existing identifiers are listed below.

**!c:** *Type for configurations of a node*

$N!c$  is the type for configurations of the node  $N$ . Elements of this type are assignments of state and flow variables that satisfy assertions of the node. Variables of the node and sub-node are accessible using the dot (.) notation.

*Example :* Consider the following ALTARICA system where a node  $B$  embeds two subnodes of kind  $A$ :

```
node A
  state x : bool
edon

node B
  state y : bool
  sub a : A[2]
edon
```

The following lines give valid MEC 5 equations:

```
arc>eval
eval>R1(x : B!c) := (x.a[0].x = x.y);
eval>R2(x : B!c) := <z : A!c> (x.a[0] = z and z.x = x.y);
eval>R3(x : B!c) := R1(x) != R2(x);
eval>EOF
```



```

R1 (x : B!c) : 4 elements / 3 nodes
R2 (x : B!c) : 4 elements / 3 nodes
R3 (x : B!c) : empty
arc>

```

**!epsilon:** *Epsilon event (\$)*

$N!$ epsilon is a singleton subset of  $N!$ ev (see below). It contains the event of  $N$  composed with only elementary epsilon events.

*Example :*

```

arc>eval
node A
  state x : bool;
  event a;
  trans true |- a ->;
edon
node B
  sub a : A;
  state x : bool;
  event b;
  trans true |- b -> ;
edon
arc>EOF
arc>eval
eval>R1(e : B!ev) := true;
eval>R2(e : B!ev) := e. = '$';
eval>R3(e : B!ev) := B!epsilon(e);
eval>EOF
R1 (e : B!ev) : complete (4 elements)
R2 (e : B!ev) : 2 elements / 2 nodes
R3 (e : B!ev) : 1 elements / 3 nodes
arc>show R1 R2 R3
R1 contains :
e. in {b, $}, e.a. in {a, $}

R2 contains :
e. = $, e.a. in {a, $}

R3 contains :
e. = $, e.a. = $
arc>

```

**!ev:** *Type for events of a node*

$N!$ ev is the type for events of the node  $N$ . An element of  $N!$ ev is a vector built with:

- labels of local events of  $N$
- events of subnodes

The dot notation can be used to access label of events and events of sub-nodes in the following way. Assume  $x$  is a variable of type  $N!$ ev then:

- $x$ . is a variable that refers a local events of  $N$ . The value of this variable can be compared with actual labels of local events of  $N$ .
- $x.sn$  is a variable of type  $M!$ ev where  $M$  is the type of sub-node  $sn$ .
- $x.sn$ . is a variable that refers to local events of the sub-node  $sn$ .

Note that two variables, say  $e_1$  and  $e_2$ , with types respectively,  $M!$ ev and  $N!$ ev are not comparable. However  $e_1$ . and  $e_2$ . can be compared if sets of labels are equal.

*Example :*

```

arc>eval
node A
  state x : bool;

```

```

    event a, b, c;
    trans true |- a, b, c ->;
edon

node B
  state x : bool;
  event b, c, d;
  trans true |- b, c, d ->;
edon

node C
  event b, c, d;
  trans true |- b, c, d ->;
edon
eval>EOF
arc>eval
eval>R1(e1 : A!ev, e2 : B!ev) := e1 = e2;
eval>EOF
<eval>:1: error: bad expression type.
<eval>:1: error: getting type 'B!ev'.
<eval>:1: error: instead of 'A!ev'.
arc>eval
eval>R2(e1 : B!ev, e2 : C!ev) := e1. = e2.;
eval>EOF
R2 (e1 : B!ev, e2 : C!ev) : 4 elements / 6 nodes
arc>

```

**!init:** *Set of initial configurations*

$N!init$  is a subset of  $N!c$  (i.e. assertions hold). The set contains configurations that respect the initial condition specified by the `init` clauses of the node (and sub-nodes).

**!move:** *Unlabelled relation transition*

$N!move$  is a subset of  $N!sc \times N!sc$ . This relation encodes permitted moves between two assignments of variables. No pre-/post-condition is applied.

**!reach:** *Set of reachable configurations*

$N!reach$  is a subset of  $N!c$  and contains configurations that are reachable from initial configurations (see `!init` relation).

**!sc:** *Type for assignments of variables of node*

Objects of the type  $N!sc$  are any assignment of variables of  $N$ . The difference between  $N!sc$  and  $N!c$  is that the elements of the latter are constrained by assertions of the node.

*Example :*

```

arc>eval
eval>node A
  flow f : bool;
  state s : bool;
  assert s != f;
edon
eval>EOF
arc>eval
eval>R(c : A!c) := true;
eval>EOF
R (c : A!c) : 2 elements / 3 nodes
arc>show R
R contains :
c.f = true, c.s = false
c.f = false, c.s = true

arc>eval

```

```

eval>R(v : A!sc) := true;
eval>EOF

R (v : A!sc) : complete (4 elements)
arc>show R
R contains :
v.f in bool, v.s in bool

arc>

```

**!st:** *Relation transition without assertions*

$N!st$  is a subset of  $N!sc \times N!ev \times N!sc$ . This is a transition relation of the node  $N$  between assignments permitted by transitions. Source and target assignments does not necessarily satisfy assertions.

**!t:** *Relation transition between configurations*

$N!t$ , a subset of  $N!c \times N!ev \times N!c$ , the transition relation of the node  $N$ . Source and target configurations are not necessarily reachable from the initial state.

**!vs:** *Assignment of state variables that can satisfy assertions*

$N!vs$  is a subset of  $N!sc$ . An assignment in  $N!vs$  is built from a valid configuration in  $N!c$  by making flow variables free.

**!vsc:** *Transitions between valid state variables assignments*

$N!vsc$  is a subset of  $N!sc \times N!ev \times N!sc$ .  $N!vsc$  is the projection of  $N!t$  on valid assignments of state variables i.e.  $N!vs$ .

## 6 Stochastic simulation

ALTARICA is a language used to describe discrete event systems. Nowadays ALTARICA remains untimed and even if the language allows non-determinism on events and flows, no concept of randomness, in the sense of stochastic process, has been soundly designed to match with the original ALTARICA semantics (see H. Soueidan’s thesis for a try on a restricted version of the language [hs09], page 69).

In the context of risk assessment it is an actual need to be able to evaluate some measures (e.g. mean time to failure of the mission) according to known failure rates of elementary components. Many models are available to achieve this task: fault trees, Markovian processes, stochastic Petri nets, ...

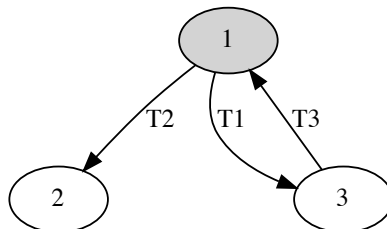
Since the release 1.6, ARC integrates a stochastic simulator for ALTARICA models that are augmented with *ad hoc* `extern` clauses. The syntax of these clauses is presented in Section 6.4 [Clauses for stochastic simulation], page 57. The simulator does not accept any ALTARICA model. The constraints that must be fulfilled by input models are given in Section 6.2 [Prerequisites on models], page 56.

The proposed extension of ALTARICA yields a model closed to generalized stochastic Petri nets ([GSPN], page 69). In the next section we give an overview of extensions and an informal description of their semantics.

### 6.1 Stochastically Timed ALTARICA

The extension of our model attaches delays to transitions. A delay is either constant (positive or null) or randomly determined according to some stochastic law. Let us assume for a while that delays are constants. Let  $d(T)$  be the delay attached to a transition  $T$ . Now assume that at instant  $t$ , the transition  $T$  is enabled; then it will be triggered after  $d(T)$  units of time. If  $T$  is disabled before  $t + d(T)$ , then the transition is not triggered.

Consider the following example where transitions  $T_1$  and  $T_2$  are in conflict from state 1; both are enabled at the same time. Now assume that  $d(T_1) < d(T_2)$ . After  $d(T_1)$  units of time,  $T_1$  is triggered and  $T_3$  is enabled while  $T_1$  and  $T_2$  are disabled. Then, after  $d(T_3)$ ,  $T_1$  and  $T_2$  are again enabled and the execution continues as previously with  $T_1$ .



The semantics of the model is a restriction of behaviors allowed by the standard semantics: here only an alternation of  $T_1$  and  $T_3$  is possible while in standard ALTARICA, the transition  $T_2$  can be triggered from state 1.

In the previous example, when  $T_2$  is disabled by the triggering of  $T_1$ , its delay is reset. Now assume that  $d(T_2)$  is related to the wear of the component and that in the state 1 the component is in use (e.g. in a production mode). Then, when  $T_2$  is enabled one more time,  $d(T_2)$  should not be the same. Our extension permits to specify that remaining delays have to be kept between

two activations. With this memorization mechanism, unless  $d(T_1) = 0$ , the transition  $T_2$  should be eventually triggered after one or more loops  $T_1 \rightarrow T_3 \rightarrow T_1$ .

In the case where  $d(T_1) = d(T_2)$ , the delay does not determine which transition is triggered. The transition is selected according to 3 policies:

- **Explicit priority.** A priority level (a natural number) is attached to each event of the model. Then the transition with the highest priority is chosen.
- **Random choice.** A weight is attached to each transition in a conflict; then a random number is drawn to select the transition according to the specified weights.
- **Internal ordering.** A transition is selected according to the internal data-structure of the tool.

The two first policies are not very relevant if they are applied to transitions with stochastic delays. Actually the probability that two transitions are enabled at the same time and have the same random delay is negligible with respect to the whole simulation.

The last policy should not be used with deterministic delays. Indeed the triggered transition is selected according to the implementation of the simulation algorithm. Choosing a transition *a priori* creates a bias in statistical results of the simulation. This bias can be correctly interpreted by the user if this latter knows the selected transition but this is not the case with this policy.

The data used for stochastic simulation are interpreted according to the *flat* semantics of the model. No composition operator has been formally designed to yield a compositional stochastic semantics and the tool must be able to associate to each **global** event one and only one probability law. Let  $\langle c, e_1, \dots, e_n \rangle$  be a synchronization vector of a compound node from which  $c$  is the event. Then, if a law is associated to  $c$  then it becomes the law associated to the vector. Else, only one  $e_i$  can be associated with a law, which becomes the law of the vector. The reader should remember that implicit synchronizations exist along the hierarchy which can yield vectors that not respect the previous constraint. To tackle this drawback,

- either the **public** attribute is used to avoid implicit synchronization of asynchronous events
- or an explicit law is associated to a control event in the compound node.

## 6.2 Pre-requisites on models

Stochastic simulation can not be applied to all ALTARICA models. We have restricted computations to models that satisfy following rules:

- **Determinism.** The model must have only one initial state and, given a source configuration, a (macro-)transition must have only one target.
- **Intrinsic flow orientation.** The tools tries to determine if flow variables only depend on state variables. If this is the case an implicit orientation of flows exists which improves simulation algorithm. Assertions of the model are examined to identify functional dependencies between variables. To be accepted, the tool must be able to associate to each flow variable such dependency without creating a cycle.
- **No free flow variable.** All flow variables must be constrained by an assertion. A free variable introduces unwanted non-determinism.
- **Unicity of laws for global events.** As mentioned in previous section, the tool must be able to associate to each global event a unique probability law.

## 6.3 Syntax of extern clauses

In versions that precedes ARC 1.6, the **extern** field of nodes was completely permissive. Since 1.6, the syntax of **extern** declarations has been specialized with the rules given below. All the rules are not currently used but we kept them to ensure some compatibility with other tools.

The **extern** field of an ALTARICA node is a list of *extern-decl* separated by semi-colons (;). An *extern-decl* has two forms given below. *identifier* is an ALTARICA identifier.

*extern-decl* ::= *identifier extern-term = extern-term*  
*extern-decl* ::= *identifier extern-term*

Then an *extern-term* is a generic sentence among:

*extern-term* ::= **true**  
*extern-term* ::= **false**  
*extern-term* ::= *signed-integer*  
*extern-term* ::= *real-number*  
*extern-term* ::= *string*  
*extern-term* ::= *identifier-path*  
*extern-term* ::= *identifier* ( (*extern-term*,)\* *extern-term* )  
*extern-term* ::= { (*extern-term*,)\* *extern-term* }  
*extern-term* ::= < **flow** *identifier-path* >  
*extern-term* ::= < **state** *identifier-path* >  
*extern-term* ::= < **event** *identifier-path* >  
*extern-term* ::= < **sub** *identifier-path* >  
*extern-term* ::= < **local** *identifier-path* >  
*extern-term* ::= < **global** *identifier-path* >  
*extern-term* ::= < **term** ( *expression* ) >

where *identifier-path* is a comma-separated list of identifiers and *expression* is an ALTARICA expression.

## 6.4 Clauses for stochastic simulation

The generic syntax given in previous section permits to describe extensions related to stochastic data.

### 6.4.1 Parameters

Parameters are identified values that can be used as argument of probabilistic laws. The syntax for the declaration of parameters is:

**parameter** *identifier-path* = *param*

where *param* is;

*param* ::= *identifier-path*  
*param* ::= *real*  
*param* ::= **uniform** ( *param*, *param* )  
*param* ::= **lognormal** ( *param*, *param* )  
*param* ::= **normal** ( *param*, *param* )

The last three forms are used to specify random parameters. Before the simulation the value of parameters are drawn according to specified laws. ARC permits the measure of effects of uncertainties on parameters using several simulations (See [Section 3.7 \[Stochastic simulation: sas\]](#), page 35).

Following examples illustrate the use of these clauses. In the first the reader should notice that this clauses can be used out of a node at the top-level scope of the model. This way global parameters can be declared; here we define a global failure rate **THE\_LAMBA** that can be used every where a parameter is allowed.

```

extern parameter THE_LAMBA = 100;

node A
  extern
    parameter lambda = THE_LAMBA;
    parameter frate = lambda;
  edon

```

Below, in node B, the failure rate `frate` of the subnode `a` is redefined (it was `THE_LAMBA`) to a value local to node B.

```

node B
  sub
    a : A;
  extern
    parameter mylambda = 1000;
    parameter a.frate = mylambda;
  edon

```

In this last example, the failure rate of a subnode `b`, is used to redefine the one of another subnode `a`.

```

node C
  sub
    a : A;
    b : B;
  extern
    parameter a.frate = b.mylambda;
    parameter mylambda = b.a.frate;
  edon

```

### 6.4.2 Laws

Probabilistic laws can be attached to a single event:

```

law event = probability-law
  where event ::= < event identifier-path >

```

or to a set of events:

```

law event-set = probability-law
  where event-set ::= { (event,)* event }

```

*Example 1:* The following example is a repairable component described in AltaRica. The repair is not available just after the failure of the component. Before that it must wait that maintainers are free. We attach exponential laws to `failure` and `repair` events while the arrival of the maintenance team is an immediate transition.

```

extern parameter LAMBDA = 1e-3;
extern parameter MU = 1e-2;

node Component
  state
    mode : ok, wait, maintenance ; init mode := ok;
  event
    failure, start_maintenance, repair : public;
  trans
    mode = ok |- failure -> mode := wait;
    mode = wait |- start_maintenance -> mode := maintenance;
    mode = maintenance |- repair -> mode := ok;
  extern
    law <event failure> = exponential (LAMBDA);
    law <event start_maintenance> = dirac (0);
    law <event repair> = exponential (MU);
  edon

```

ARC accepts many laws as shown in the following table. They are described more in detail in [Appendix B \[Probabilistic laws\]](#), page 79.

<b>Law</b>	<b>Reference</b>
<code>dirac(<math>\delta</math>)</code>	Section B.1 [Dirac's law], page 79
<code>empiric(<math>b_1, \dots, b_{12}</math>)</code>	see Section B.2 [Empiric law], page 79
<code>erlang(<math>m, \beta</math>)</code>	see Section B.3 [Erlang's law], page 79
<code>exponential(<math>\lambda</math>)</code>	see Section B.5 [Exponential law], page 80
<code>exponential_wow(<math>\lambda, \delta_1, \dots, \delta_{12}</math>)</code>	see Section B.6 [Exponential law + Wait On Weather delays], page 80
<code>gen_erlang(<math>\lambda_1, \dots, \lambda_k</math>)</code>	see Section B.4 [Generalized Erlang's law], page 80
<code>ifa(<math>\delta, t_0</math>)</code>	see Section B.8 [Instants Fixed in Advance], page 81
<code>ipa(<math>\delta</math>)</code>	see Section B.7 [Instants Provided in Advance], page 81
<code>nlog(<math>m, q</math>)</code>	see Section B.9 [Log Normal law], page 81
<code>optional(<math>c_1, l_1, \dots, c_n, l_n</math>)</code>	see Section B.10 [Optional laws], page 82
<code>triangle(<math>a, b, h</math>)</code>	see Section B.11 [Triangular law], page 82
<code>truncated_weibull(<math>m, \beta, \alpha</math>)</code>	see Section B.14 [Truncated Weibull's law], page 84
<code>uniform(<math>a, b</math>)</code>	see Section B.12 [Uniform law], page 83
<code>weibull(<math>m, \beta</math>)</code>	see Section B.13 [Weibull's law], page 83

### 6.4.3 Observers

Observers are just expressions over variables of the model. ARC distinguishes two kind of observers *properties* and *predicates*. Formally, the only difference is the type of the observed expression; in the first case it is a numerical value while in the second case it is a Boolean one.

The kind of observer only influences statistics that are displayed at the end of the simulation (see [Section 6.5 \[Example\]](#), page 60).

The syntax for observers is:

```
property identifier = <term ( expression ) >
predicate identifier = <term ( expression ) >
```

### 6.4.4 Memorization of delays

It is possible to indicate that the probabilistic law of an event uses a memory. The syntax is the following:

```
preemptible event
preemptible event-set
```

If an event is marked as `preemptible`, each time it is disabled, the remaining time until its triggering is memorized.

A synchronization vector is marked as `preemptible` if one of its components has the flag.

### 6.4.5 Priority

Priority levels can be assigned to events using the following syntax:

```
priority event = integer
priority event-set = integer
```

If some events are in conflict, the one with the highest level of priority will be triggered.

### 6.4.6 Random choices

Failures on demand can be described using `buckets`. A bucket gathers several events that are usually associated with deterministic laws. If events belonging to a bucket are enabled at the



same instant, the triggered one is chosen randomly according to specified weights. The syntax for describing a bucket is the following:

```
bucket identifier = { (event, param,)* event }
```

The *identifier* is used to identify the bucket and is displayed in results. Each *event* is followed by its weight *param*. Weights are constant parameters and their sum must be  $< 1$ . The weight of the last event is the complement to 1.

*Example 2:* Following example is a component with three failure modes. It waits for **start** command but it also may fail at start-up:

- either it enter a mode `fail_stuck` where it ignores the changes of its input `i`
- or it enters `fail_down` mode where it always sends `false` on its output `o`.

If the component successes to start it enters a `running` state from which it may also fail with a failure rate `LAMBDA`.

A bucket `fail_on_start` is created to gather events `start`, `fail_stuck` and `fail_down` with respective weights 0.6, 0.2 and 0.2.

```
node Component
  flow i, o : bool;
  state
    mode : idle, run, fail_down, fail_stuck ;
    stuck_value : bool;
  init
    mode := idle, stuck_value := false;
  event
    start, stop, fail_stuck, fail_down, failure : public;
  trans
    mode = idle |- fail_stuck -> mode := fail_stuck, stuck_value := i;
    mode = idle |- fail_down -> mode := fail_down;
    mode = idle |- start -> mode := run;
    mode = run |- failure -> mode := fail_down;
    mode = run |- stop -> mode := idle;
  assert
    case
      mode = run : o = i,
      mode = fail_stuck : o = stuck_value,
      else o = false
    ;
  extern
    parameter LAMBDA = 0.001;
    law <event start>, <event stop>,
      <event fail_stuck>, <event fail_down> = dirac (0.0);
    law <event failure> = exponential (LAMBDA);
    bucket fail_on_start =
      <event start>, 0.6,
      <event fail_stuck>, 0.2,
      <event fail_down> ;
  edon
```

## 6.5 Example

The stochastic simulator is invoked with [\[sas command\]](#), page 35. The following example is a system with two repairable components and one maintainer i.e. at most one machine is repaired at each instant.

Four observers are declared in the top-level node to observe combination of failures of machines.

```
extern parameter LAMBDA = 1e-4;
extern parameter MU = 4.1666e-2;

node Component
```

```

state
  mode : ok, wait, maintenance ; init mode := ok;
event
  failure, start_maintenance, repair : public;
trans
  mode = ok |- failure -> mode := wait;
  mode = wait |- start_maintenance -> mode := maintenance;
  mode = maintenance |- repair -> mode := ok;
extern
  law <event failure> = exponential (LAMBDA);
  law <event start_maintenance> = dirac (0);
  law <event repair> = exponential (MU);
edon

node Maintainer
  state s : idle, work ; init s := idle;
  event
    start_repair, end_repair;
  trans
    s = idle |- start_repair -> s:= work;
    s = work |- end_repair -> s := idle;
edon

node System
  sub
    maintainer : Maintainer;
    machine : Component[2];
  sync
    <maintainer.start_repair, machine[0].start_maintenance>;
    <maintainer.start_repair, machine[1].start_maintenance>;
    <maintainer.end_repair, machine[0].repair>;
    <maintainer.end_repair, machine[1].repair>;
  extern
    predicate all_ok = <term (machine[0].mode = ok & machine[1].mode = ok)>;
    predicate all_down = <term (machine[0].mode != ok & machine[1].mode != ok)>;
    predicate m1_down = <term (machine[0].mode != ok & machine[1].mode = ok)>;
    predicate m2_down = <term (machine[0].mode = ok & machine[1].mode != ok)>;
edon

sas --duration=8760 --nb-stories=8000 --seed=123456781 System

```

The last line of the file invokes `sas`. The duration is set to 8760 hours (one year). The failure rate of component is 1 failure every 10000 hours and the machine are repaired in less than 24 hours.

```

*** ACTION FREQUENCIES
      NAME                MEAN          STDDEV          CONF.
<machine[0].start_maintenance, maintainer.start_repair>      8.68125E-01
  9.17382E-01          1.68209E-02
<machine[1].start_maintenance, maintainer.start_repair>      8.65500E-01
  9.38627E-01          1.72104E-02
<machine[0].repair, maintainer.end_repair>          8.66625E-01          9.16074E-01
  1.67969E-02
<machine[1].repair, maintainer.end_repair>          8.64000E-01          9.37877E-01
  1.71967E-02
<machine[1].failure, machine[1].failure>          8.65500E-01          9.38627E-01
  1.72104E-02
<machine[0].failure, machine[0].failure>          8.68125E-01          9.17382E-01
  1.68209E-02

*** CUMULATED TIME WITH EXPECTED VALUE
      NAME                MEAN          STDDEV          CONF.
      all_ok              8.71909E+03      4.42254E+01      8.10905E-01
      all_down            1.21229E-01      2.32423E+00      4.26165E-02
      m1_down             2.06645E+01      3.11001E+01      5.70245E-01

```



m2_down	2.01222E+01	3.10510E+01	5.69344E-01
*** NUMBER OF OCCURRENCES			
NAME	MEAN	STDDEV	CONF.
all_ok	2.72538E+00	1.30160E+00	2.38659E-02
all_down	5.25000E-03	7.22710E-02	1.32514E-03
m1_down	8.68125E-01	9.17382E-01	1.68209E-02
m2_down	8.65500E-01	9.38627E-01	1.72104E-02
*** FIRST OCCURRENCES			
NAME	MEAN	STDDEV	CONF.
CENSURED			
all_ok	0.00000E+00	0.00000E+00	0.00000E+00
0			
all_down	4.47732E+03	2.53250E+03	6.40867E+02
7958			
m1_down	3.79482E+03	2.49039E+03	5.97274E+01
3324			
m2_down	3.79306E+03	2.49124E+03	6.02657E+01
3404			

## 7 Altarica Studio

Current release of ARC is accompanied with the prototype of a small graphical user interface (GUI) humbly called ALTARICA STUDIO (or AS for short). The aim of AS is to help AltaRica users to validate their models given them rapid access to elementary informations related to components they describe in their AltaRica files.

### 7.1 Validation tools

The main window (see [Figure 7.1](#)) is divided in 4 parts:

1. A menu that proposes to load a file or to quit AS.
2. A toolbar with two buttons:  used to refresh displayed data when ARC console is used (see [Figure 7.7](#)) and  that spawn the graphical simulator (see [Section 7.2 \[Simulator\]](#), [page 66](#)) for the currently selected node.
3. The hierarchy of currently loaded AltaRica nodes.
4. A set of tabs that give access to predefined ARC commands.

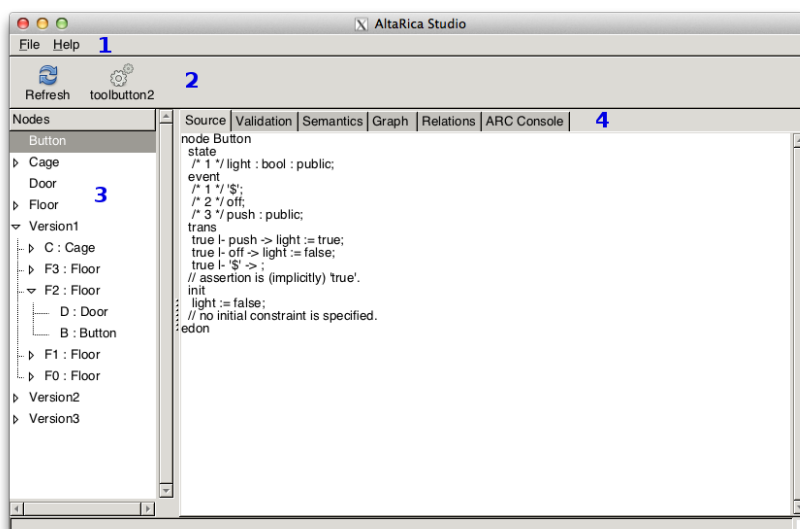


Figure 7.1: Main window of Altarica Studio

By default the *Source* tab is selected. This tab displays the code of the constraint automaton that represents the flat semantics of the selected node.

The second tab, called *Validation*, request ARC to apply [[validate command](#)], [page 27](#) to the selected node and then displays the result. For high-level nodes (e.g. a *Main*) this computation can require several minutes; meanwhile AS is frozen.

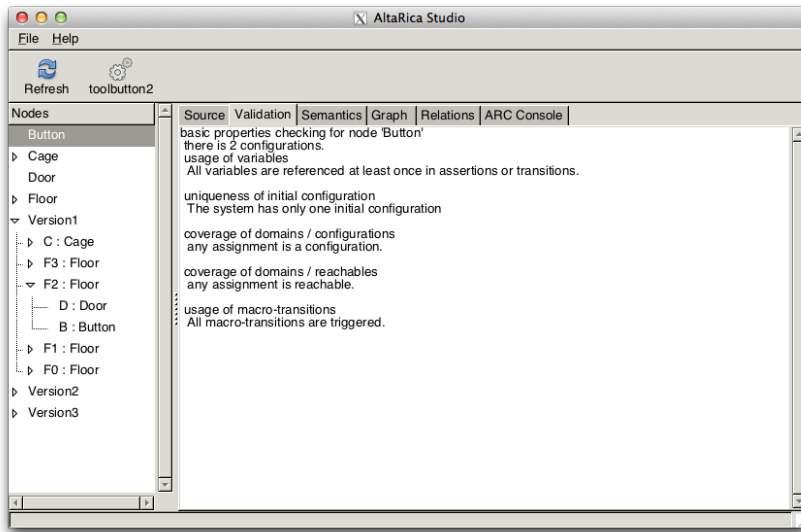


Figure 7.2: Results of `validate` command are displayed in *Validation* tab.

Next tab, *Semantics*, displays the state-graph of the selected node. Since such graph is human-readable for a small number of states, AS disables the display of semantics beyond one hundred of states and a warning window appears if the state graph could have more than 15 states. To take this decision AS does not actually compute the semantics; it just estimate the worse case by computing the cardinality of the cartesian product of domains.

The purpose of this tabs is first, and foremost, to ease the validation of smallest components of the model. With such automata the user can easily check if at least leaf nodes of the system are valid.

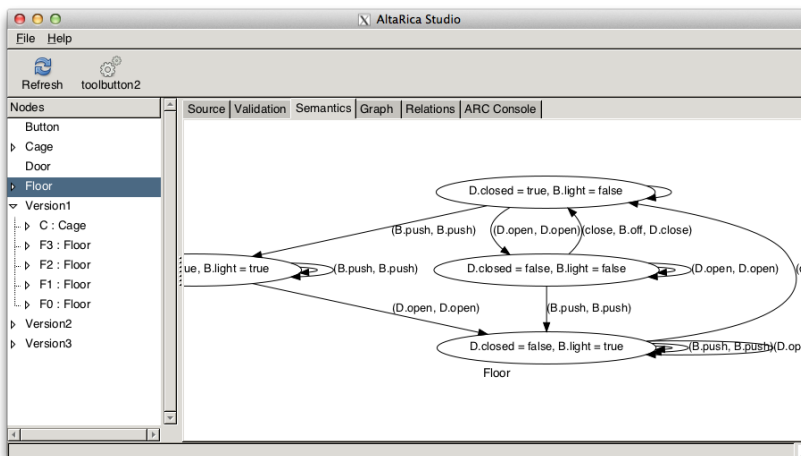


Figure 7.3: State-graph of the selected node.

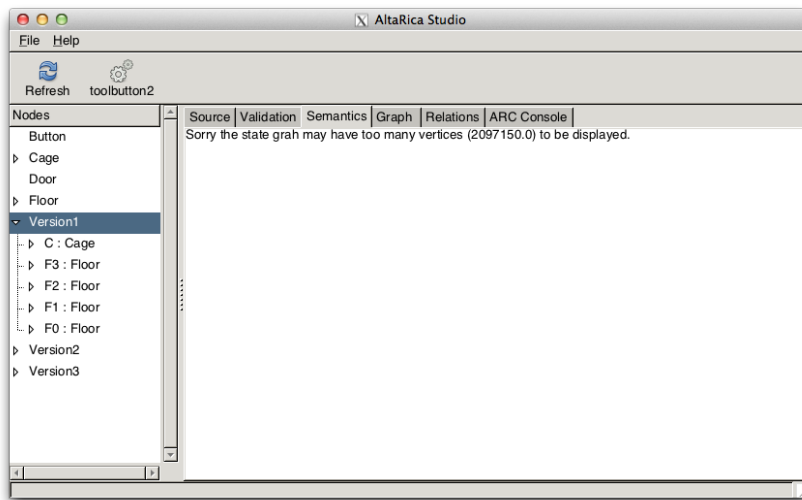


Figure 7.4: When state space is tool large, the graph is not displayed.

Fourth tab permits to display the graph of ACHECK commands that have an outputs in DOT format (e.g., `mode` or `dottrace`). It suffices to type the command without any `with.. do .. done` context.

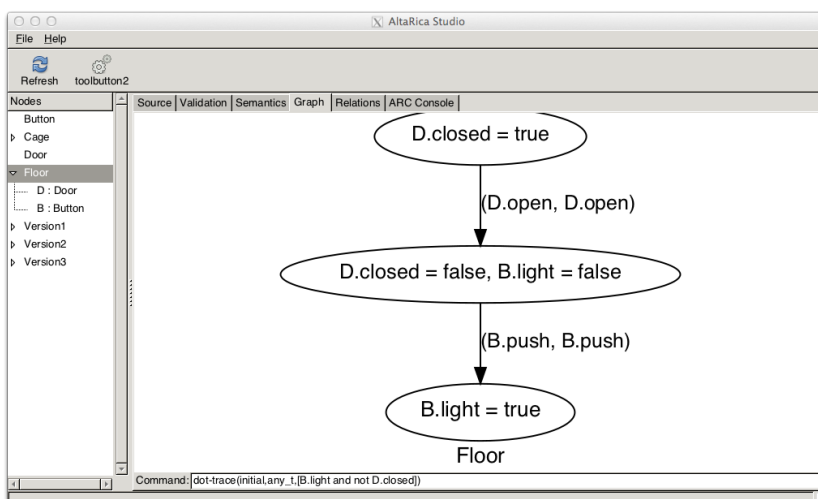


Figure 7.5: Graph tab displays graph for DOT compliant commands.

Fifth tab gathers all computed relations, either with MEC 5 or ACHECK specifications. Lines can be sorted by clicking on the head of the column *Relations*.

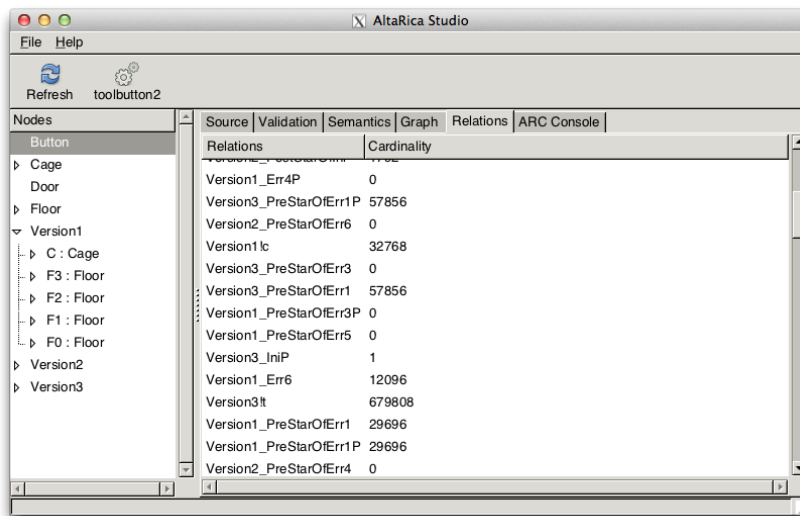


Figure 7.6: Name and cardinality of relations already computed.

The last tab permits to interact with the ARC process spawned by AS. The user can enter ARC commands and results are displayed into the window. Up and down arrows can be used to visit history of commands.

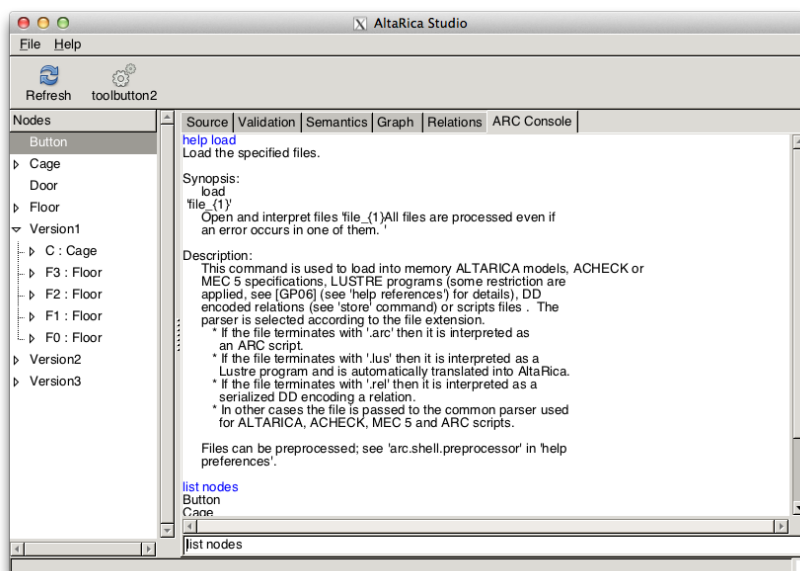



Figure 7.7: Direct interaction with ARC.

## 7.2 Simulator

When a node is selected and the tool button  is pressed, a step-by-step graphical simulator is started for that node. Several simulators for a same or other nodes can be opened. The main window of the simulator looks like on the figure [Figure 7.8](#).

The window is divided in several parts:

1. Two arrows permit to go through the simulation stack. An history of visited states is

maintained. With these arrows, the user can go forward or backward. These movements are independent of state changes i.e by triggering transitions or by direct jump to one state using the mouse.

2. A frame labelled *Configurations* displays assignments of variables. Variables are presented in a tree that can be unfolded. In case of non-determinism several configurations are displayed. In this situation, several columns are displayed and the user selects a configuration by clicking on the **X** of the corresponding column.
3. In this area the part of the state-graph explored so far is displayed. Assignment of variables is not displayed but it can be viewed in *Configurations* frame. The current state is filled in red. Clicking on a state with **CTRL** key pressed selects the vertex as the new current configuration. Corresponding assignment is updated in *Configurations* frame and available transitions in *Transitions* frame.
4. *Transitions* frame lists transitions that can be triggered from the current configuration. Transitions are gathered according to their labelling event. Under each event is displayed the guard and assignments of the transitions. Double-clicking on the line containing the guard triggers the transition. If the set of target configurations is a singleton then it is set as the current one else, the user has to select one among those proposed in *Configurations* frame.

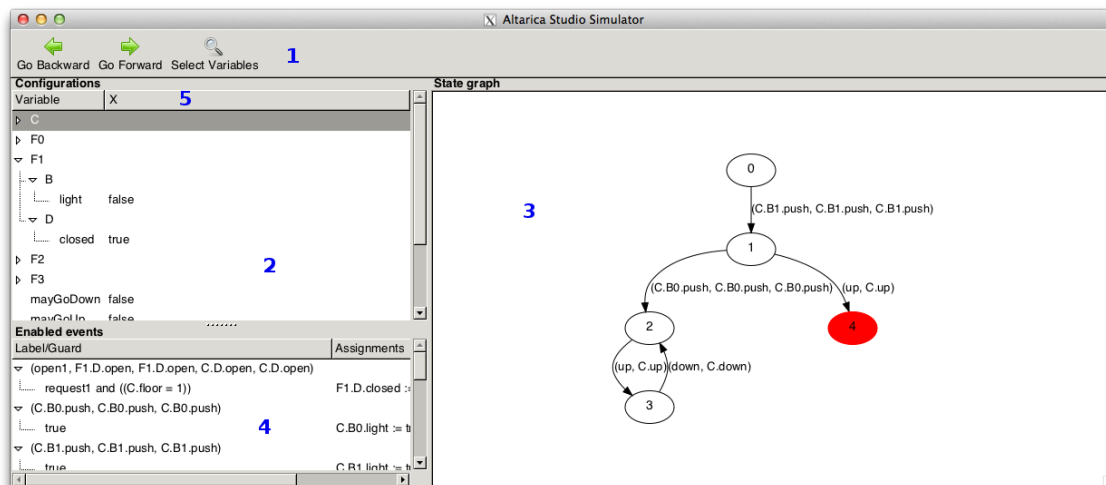


Figure 7.8: The graphical simulator





## 8 References

- [AC88] A. Arnold and P. Crubillé. A linear algorithm to solve fixed-point equations on transition systems. In *Information Processing Letters*, vol 29, Issue 2, Elsevier, 1988.
- [AGP99] A. Arnold, A. Griffault, G. Point and A. Rauzy. *The AltaRica formalism for describing concurrent systems*. *Fundam. Inf.* 40, issue 2-3, pages 109-124, IOS Press, 2000.
- [AKPV11] A. Griffault, F. Kuntz, G. Point and A. Vincent. Symbolic computation of minimal cuts for AltaRica models. Research Report n° 1456-11. LaBRI – Université Bordeaux I, Sep. 2011.
- [ALTA] Web site of the *AltaRica Project*: <http://altarica.labri.fr/>.
- [AV03] A. Vincent. *Conception et réalisation d'un vérificateur de modèles AltaRica*. Thèse de l'Université Bordeaux I, 2003.
- [CR97] M.-M. Corsini and A. Rauzy. Toupie: the mu-calculus over Finite Domains as Constraint Language. *Journal of Automated Reasoning*, Volume 19, Number 2, pages 143-171, Springer, 1997.
- [DOT] Graphviz - Graph Visualization Software. <http://www.graphviz.org/>, 2009.
- [EH86] E.A. Emerson and J. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time. *J. ACM* 33, pp 151-178, 1986.
- [AL15] G. Audemard and L. Simon. *The Glucose SAT Solver*. 2015
- [GP00] G. Point. *AltaRica: Contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement*. Thèse de l'Université Bordeaux I, 2000.
- [GP06] A. Griffault and G. Point. *On the partial translation of Lustre programs into the AltaRica language and vice versa*, Research Report n° 1415-06, LaBRI, Nov. 2006.
- [MM94] M. Dam. CTL\* and ECTL\* as fragments of the modal  $\mu$ -calculus. *Theoretical Computer Science* 126, pp. 77-96, Elsevier, 1994
- [RL] Web site of the *GNU Readline Library*.
- [HS09] H. Soueidan. *Discrete event modeling and analysis for systems biology models*. Thèse de l'Université de Bordeaux I, 2009.
- [GSPN] M. Ajmone Marson, G. Balbo, G. Conte, S. Donatelli and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, Inc., 1994.
- [MW89] M. R. Wingo. The left-truncated Weibull distribution: theory and computation. *Statistical Papers* 30, 39–48, Springer Verlag, 1989
- [GLU] The Glucose Solver. <http://www.labri.fr/perso/lsimon/glucose/>. 2017
- [RB98] R. Brown. Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem. *Communications of the ACM*. Vol. 31 N. 10. October, 1998.



## Appendix A User preferences

This appendix gathers user-definable preferences that modify the behavior of ARC. To modify or simply display the current value of preferences use `[set command]`, page 17.

### A.1 Shell

This section lists preferences related to the command-line of ARC.

`arc.shell.check-card-abort` :

Abort the program if the check-card command fails.

Possible values: Booleans

Default value: `false`

`arc.shell.history_file` :

The name of the file use to save the history of commands used during the last ARC session.

Possible values: Any valid file name

Default value: `~/arc_history`

`arc.shell.preprocessor` :

ARC permits to preprocess ALTARICA, MEC 5 or ACHECK files. One can associate to filename extensions (except `.arc`, `.lus` and `.rel`) a program used as a preprocessor. To specify that files terminating with the extension `.ext` must be preprocessed using a specific program, set the preference `arc.shell.preprocessor.ext.command` to the a string that contains the name of the executable (that must accessible via the `PATH` environment variable). A `%s` in the specified string indicates the position of the file to process. If no `%s` is found then the filename is just concatenated with the specified command and a separating white space.

Additional arguments can be appended to the command using the preferences `arc.shell.preprocessor.ext.args`. Such additional arguments are added each time the preprocessor is called.

If ARC can not associate a preprocessor to an extension it looks for preferences for `arc.shell.preprocessor.default.command` and `arc.shell.preprocessor.default.args`. If these latters are not set the file is parsed as-is.

Note that the parser handles `#line` indications generated by certain preprocessors.

Examples: With the following setting, any file `F` with the extension `.php` are pre-processed with the command `php F 5`.

```
set arc.shell.preprocessor.php.command "php"
set arc.shell.preprocessor.php.args "5"
```

Possible values: String

Default value: none

`arc.shell.prompt.0` :

The level 0 prompt.

Possible values: Any character string without newline

Default value: `arc>`

`arc.shell.prompt.1` :

The level 1 prompt.

Possible values: Any character string without newline

Default value: `.`

`arc.shell.verbose` :

Enable/disable (verbose) informative messages emitted by the ARC shell.

Possible values: Booleans

Default value: `true`

## A.2 Acheck

This section list preferences related to ACHECK specifications.

`acheck.cleanup-semantic` :

Enable/disable the deletion of semantics after computation.

Possible values: Booleans

Default value: `false`

`acheck.dot-diff-state-mode` :

When ACHECK uses Decision Diagrams engine, this option specifies how DOT command outputs configurations. Four modes are available:

- `none`: configurations are displayed as-is.
- `post`: project configurations on variables that are changed by a step forward (i.e a 'post'). Variables that keep their value after any outgoing transition are not displayed.
- `pre`: only variables that have a value different from at least on predecessor of the configuration are kept and displayed.
- `both`: apply `post` and `pre`.

Possible values: `none`, `post`, `pre`, `both`

Default value: `none`

`acheck.nrtest-failure-aborts` :

Aborts program if non-regression tests (`nrtest` command) fail.

Possible values: Booleans

Default value: `false`

`acheck.timers` :

Enable/disable timers for acheck computations.

Possible values: Booleans

Default value: `false`

## A.3 Mec V

This section list preferences related to MEC 5 specifications.

`mec5.timers` :

Enable/disable timers for MEC 5 computations.

Possible values: Booleans

Default value: `false`

## A.4 Translation of ALTARICA models into LUSTRE programs

Here are preferences related to the `to-lustre` command.

`translators.a2l.deterministic` :

In ALTARICA nodes an event can be used for several (macro-)transitions; this feature permits to create non-deterministic behaviors. Since non-determinism is not suitable

in the context of Lustre, this preference (if true) can be use to enforce (with an assertion) that only one of the possible (macro-)transitions is authorized.

Possible values: Booleans

Default value: `true`

**translators.a2l.enum-prefix :**

ALTARICA permits the use of symbolic values (like 'enum's in C). Each such value is translated into Lustre as a global integer constant. In order to prevent identifier conflicts a prefix (defined by this preference) is added to each symbolic value.

Possible values: Identifier

Default value: `ENUM_`

**translators.a2l.event-var-suffix :**

Prefix added to each identifiers of events.

Possible values: Identifier

Default value: `_`

**translators.a2l.flow-var-prefix :**

Prefix added to each identifiers of flow variables.

Possible values: Identifier Default value: `f_`

**translators.a2l.guard-var-prefix :**

Prefix of local variables used to store the current variable of guards.

Possible values: Identifier

Default value: `ec_`

**translators.a2l.missing-init-prefix :**

If a state variable is not initialized then a dummy parameter is added to the LUSTRE node. This dummy parameter is used to initialize the state variable. The name of the dummy parameter is the concatenation of this setting and the name of the state variable.

Possible values: Identifier

Default value: `INIT_VAL_`

**translators.a2l.noinput-name :**

If an ALTARICA node does not use any variable with an 'in' attribute then a dummy variable is created. The identifier of this variable is defined by this preference.

Possible values: Identifier

Default value: `noinput`

**translators.a2l.noreturn-name :**

If an ALTARICA node does not use any variable with an 'out' attribute then a dummy variable is created. The identifier of this variable is defined by this preference.

Possible values: Identifier

Default value: `noreturn`

**translators.a2l.parameter-prefix :**

Prefix added to each identifiers of parameters.

Possible values: Identifier

Default value: `p_`

**translators.a2l.show-sections :**

Enable/disable comments identifying parts of the produced code.

Possible values: Booleans

Default value: `true`

`translators.a21.signature-arg-prefix :`

Prefix of arguments of external functions

Possible values: Identifier

Default value: `_arg_`

`translators.a21.signature-ret-suffix :`

Suffix of the return value for external functions

Possible values: Identifier

Default value: `_return`

`translators.a21.state-var-prefix :`

Prefix added to each identifiers of state variables.

Possible values: Identifier

Default value: `s_`

`translators.a21.verbose :`

Enable/disable the display of timers for acheck computations

Possible values: Booleans

Default value: `true`

`translators.a21.warning :`

Enable/Disable the display of warnig messages.

Possible values: Booleans

Default value: `true`

## A.5 Translation of LUSTRE programs into ALTARICA models

This section lists preferences related to the translator of LUSTRE programs into ALTARICA.

`translators.l2.clock-event-name :`

All translated nodes used one event (in addition to epsilon  $\epsilon$ ). This event is used at the top-level to synchronize all nodes together.

Possible values: Identifier

Default value: `clock`

`translators.l2.initial-state-name :`

If a LUSTRE node is not purely functional i.e. it uses `pre` like operators, then the translator uses a new state variable to encode the fact that the node is initialized or not. The initial state is used to initialized the actual state variables of the LUSTRE node.

Possible values: Identifier

Default value: `_is_init`

`translators.l2.integers.as-range :`

If this preference is set to true then each occurrence of the `int` type in the LUSTRE code is replaced by a range of integers in the ALTARICA model. The bounds of the range are defined with the `translator.l2a.integers.min` and `translator.l2a.integers.max` preferences.

Possible values: Booleans

Default value: `false`

**translators.l2.integers.max :**

If `translator.l2a.integers.as-range` is true then the value specified by this preference is used as the upper bound of the range used in replacement of the `int` type.

Possible values: Integer

Default value: 10

**translators.l2.integers.min :**

If `translator.l2a.integers.as-range` is true then the value specified by this preference is used as the lower bound of the range used in replacement of the `int` type.

Possible values: Integer

Default value: -10

**translators.l2.pre-var-prefix :**

The use of the 'pre' operator in the Lustre code induces the creation of state variables in the ALTARICA node. This preference defines the common prefix for all such variables.

Possible values: Identifier

Default value: pre\_

**translators.l2.reals.add :**

Identifier of the signature of the addition between `reals`.

Possible values: Identifier

Default value: real\_add

**translators.l2.reals.div :**

Identifier of the signature of the division between `reals`.

Possible values: Identifier

Default value: real\_div

**translators.l2.reals.geq :**

Identifier of the signature of the  $\geq$  relation over 'real's.

Possible values: Identifier

Default value: real\_geq

**translators.l2.reals.gt :**

Identifier of the signature of the  $>$  relation over `reals`.

Possible values: Identifier

Default value: real\_gt

**translators.l2.reals.leq :**

Identifier of the signature of the  $\leq$  relation over `reals`.

Possible values: Identifier

Default value: real\_leq

**translators.l2.reals.lt :**

Identifier of the signature of the  $<$  relation over `reals`.

Possible values: Identifier

Default value: real\_lt

**translators.l2.reals.mod :**

Identifier of the signature of the modulo between `reals`.



Possible values: Identifier

Default value: `real_mod`

`translators.12.reals.mul` :

Identifier of the signature of the multiplication between `reals`.

Possible values: Identifier

Default value: `real_mul`

`translators.12.reals.neg` :

Identifier of the signature of the opposite of a `real` number.

Possible values: Identifier

Default value: `real_neg`

`translators.12.reals.sub` :

Identifier of the signature of the subtraction between `reals`.

Possible values: Identifier

Default value: `real_sub`

`translators.12.reals.typename` :

Identifier of the abstract type used in place of `real`.

Possible values: Identifier

Default value: `REALS`

`translators.12.reals.zero` :

Identifier of the abstract constant of type 'real' encoding zero.

Possible values: Identifier

Default value: `real_zero`

`translators.12.struct_field_prefix` :

Composite types of Lustre are replaced by structures in ALTARICA. This preference is used to name fields of such structures.

Possible values: Identifier

Default value: `_field_`

`translators.12.subnodes-prefix` :

Each node call in the Lustre code implies the creation of a sub-node in the ALTARICA node of the caller. Each sub-node is identified with a name of the form  $pref_i$  where  $pref$  is the prefix specified by this setting and  $i$  is an integer.

Possible values: Identifier

Default value: `sc_`

`translators.12.tmp_flow_var_prefix` :

The translator may create auxiliary variables in the AltaRica (e.g. to store sub-expression values). This preference sets the prefix of such variables.

Possible values: Identifier

Default value: `_f_tmp_`

`translators.12.verbose` :

Enable/disable the display of informatives messages during the translation.

Possible values: Booleans

Default value: `true`

`translators.12.warning :`

Enable/disable the display of warning messages.

Possible values: Booleans

Default value: `true`



## Appendix B Probabilistic laws

In this chapter we list probabilistic laws available for stochastic simulation. For each law we give its syntax and the algorithm use to generate random delays according to the cumulative distribution function (CDF) of the law.

Graphical representations of functions are taken from from Wikipedia website.

### B.1 Dirac's law

This law permits to attach constant delays to events. Whatever the current instant is, the event will be triggered after the specified delay  $\delta$ .

*Syntax:* `dirac( $\delta$ )`

where  $\delta$  is any valid parameter (Section 6.4.1 [Parameters], page 57).

*Random delay generation:*

- return  $\delta$

### B.2 Empiric law

The empiric law describes  $n$  equiprobable classes:  $[b_1, b_2[$ ,  $[b_2, b_3[$ ,  $\dots$ ,  $[b_n, b_{n+1}[$ . The cumulative distribution  $F(t)$  of the law is a piecewise linear function defined by:

- $F(t) = 0$  if  $t \leq b_1$
- $F(t) = \frac{1}{n}(\frac{t}{b_{i+1}-b_i} + i - 1)$  if  $b_i \leq t < b_{i+1}$  for  $i \in \{1, \dots, n\}$
- $F(t) = 1$  if  $b_{n+1} \leq t$

*Syntax:* `empiric( $b_1, \dots, b_{n+1}$ )`

where each  $b_i$  is any valid parameter (Section 6.4.1 [Parameters], page 57).

*Random delay generation:*

- let  $z$  be a random number uniformly drawn in  $[0,1]$ .
- let  $i = \lfloor n.z \rfloor + 1$
- return  $b_i + (b_{i+1} - b_i)(n.z - \lfloor n.z \rfloor)$

### B.3 Erlang's law

This law specifies that delays follow an Erlang distribution parameterized its mean  $m$  and its order  $k$ .

*Syntax:* `erlang( $m, k$ )`

where  $m$  and  $k$  are parameters (Section 6.4.1 [Parameters], page 57).

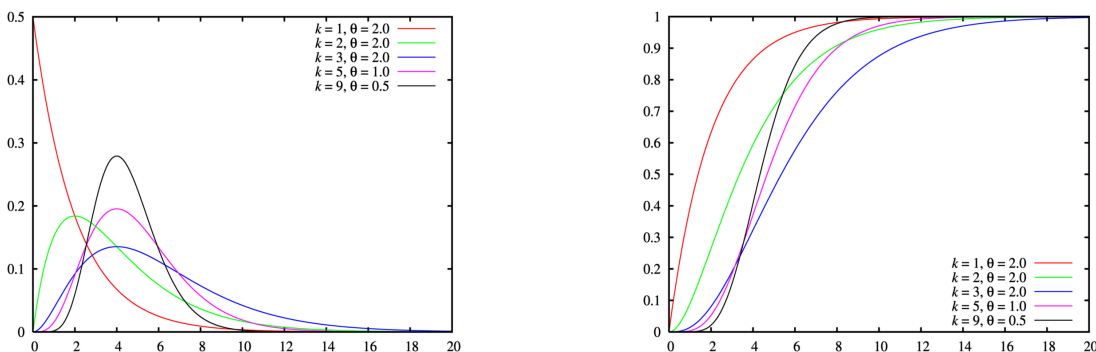


Figure B.1: PDF and CDF of Erlang's law ( $\theta = m/k$  is the *scale* of the law)

*Random delay generation:*

- let  $z_1, \dots, z_k$  be random numbers uniformly drawn in  $[0,1]$ .
- let  $\lambda = \frac{k}{m}$ .
- return  $-\frac{1}{\lambda} \ln(z_1 \times \dots \times z_k)$

## B.4 Generalized Erlang's law

While the Erlang's law represents the sum of  $k$  exponential variables with a common rate  $\lambda$ , the `gen_erlang` allows to specify different rates  $\lambda_i$ s.

*Syntax:* `gen_erlang( $\lambda_1, \dots, \lambda_k$ )`

where  $\lambda_i$  are valid parameters (Section 6.4.1 [Parameters], page 57).

*Random delay generation:*

- let  $z_1, \dots, z_k$  be random numbers uniformly drawn in  $[0,1]$ .
- return  $-\sum_{i=1}^k \sup k \frac{\ln(z_i)}{\lambda_i}$

## B.5 Exponential law

Exponential law is attached to events with a constant rate of occurrence *lambda* (i.e. the number of occurrences per time unit).

*Syntax:* `exponential( $\lambda$ )`

where  $\lambda$  is any valid parameter (Section 6.4.1 [Parameters], page 57).

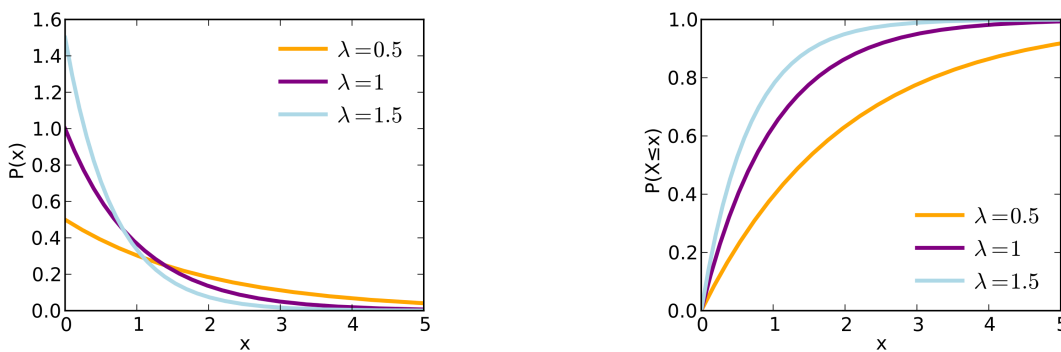


Figure B.2: PDF and CDF of exponential law for different rate  $\lambda$ .

*Random delay generation:*

- let  $z$  be a random number uniformly drawn in  $[0,1]$ .
- return  $-\frac{\ln(z)}{\lambda}$  if  $z > 0$  or 0.0 else.

## B.6 Exponential law + Wait On Weather delays

This law is applied to models for which the time unit is the hour. Its allows to add to an exponential delay a constant value that depends on the current month. Basically this law has been designed to express the additional time for maintenance operations due to weather conditions (the delay change according to current month).

*Syntax:* `exponential_wow( $\lambda, \delta_1, \dots, \delta_{12}$ )`

where:

- $\lambda$  and  $\delta_i$ s are valid parameters (Section 6.4.1 [Parameters], page 57).  $\lambda$  is the rate of the exponential law and  $\delta_i$  is the constant additional delay for the  $i$ sup *th* month.

*Random delay generation:*

- let  $z$  be a random number uniformly drawn in  $[0,1]$ .

- if  $T$  is the current time then let  $i = \lfloor T/730 \rfloor \bmod 12 + 1$ .
- return  $\delta_i - \frac{\ln(z)}{\lambda}$  if  $z > 0$  or  $\delta_i$  else.

## B.7 Instants Provided in Advance

This law is deterministic; it allows to specify delays that match a constant period  $\tau$  from the start of the simulation.

*Syntax:* `ipa( $\tau$ )`

where  $\tau \neq 0$  is any valid parameter (Section 6.4.1 [Parameters], page 57).

*Random delay generation:*

- let  $T$  be the current time
- return
  - $\tau - (T \bmod \tau)$  if  $T > \tau$
  - $\tau - T$  else

## B.8 Instants Fixed in Advance

This law is similar to `ipa`. It is deterministic and allows to specify delays that match a constant period  $\tau$  from an initial  $t_0$ . Actually `ipa( $\tau$ ) = ifa( $\tau$ , 0)`.

*Syntax:* `ifa( $\tau$ ,  $t_0$ )`

where  $\tau$  and  $t_0$  are valid parameters (Section 6.4.1 [Parameters], page 57).

*Random delay generation:*

- let  $T$  be the current time
  - $\tau - ((T - t_0) \bmod \tau)$  if  $T > t_0$
  - $t_0 - T$  else

## B.9 Log Normal law

This law allows the use of delays that follow log-normal distribution parameterized by its mean  $m$  and  $q$  its error factor to 5%. These parameters correspond to parameters  $\sigma$  and  $\mu$  of the underlying normal law as follows:  $\sigma = \ln(q)/1.645$  and  $\mu = \ln(m) - \sigma \sup 2/2$ .

*Syntax:* `nlog( $m$ ,  $q$ )` where  $m$  and  $q$  are valid parameters (Section 6.4.1 [Parameters], page 57).

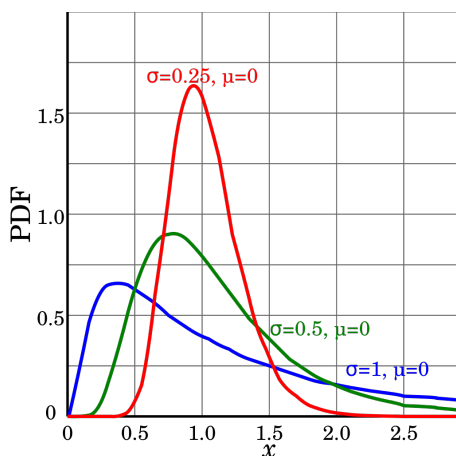


Figure B.3: PDF of log-normal law

*Random delay generation:*

- let  $z_1$  and  $z_2$  be a random numbers uniformly drawn in  $[0,1]$ .

- let  $d = \sqrt{-2 \times \ln(z_1)} \times \cos(2\pi z_2)$
- let  $\sigma = \ln(q)/1.645$  and  $\mu = \ln(m) - \sigma \sup 2/2$ .
- return  $\exp(\sigma d + \mu)$

## B.10 Optional laws

This law permits to apply laws according to some conditions. The arguments of `optional` is a list of couples  $(c_i, l_i)$  where  $c_i$  is a Boolean expression on variables and  $l_i$  is a any kind of law except an `optional` one. When the associated event is enabled, the  $c_i$ s are checked in order and if  $c_i$  is evaluated to `true` according to the current configuration of the model, then  $l_i$  is used to compute the delay applied to the event. Obviously at least one of the  $c_i$ s must be evaluated to `true` when the event is enabled.

*Syntax:* `optional`( $c_1, l_1, \dots, c_n, l_n$ )

with

$c_i ::=$  any valid *AltaRica* expression

$l_i ::=$  any not-optional law

*Random delay generation:*

- let  $i$  be the smallest integer such that  $c_i$  is `true`.
- return the random delay for law  $l_i$ .

## B.11 Triangular law

This law is used when the random delay is known to be a range  $[a, b]$  and that its most probable value is  $c \in [a, b]$ .

*Syntax:* `triangle`( $a, b, c$ )

where  $a, b$  and  $c$  are parameters such that  $a < b$  and  $a \leq c \leq b$ .

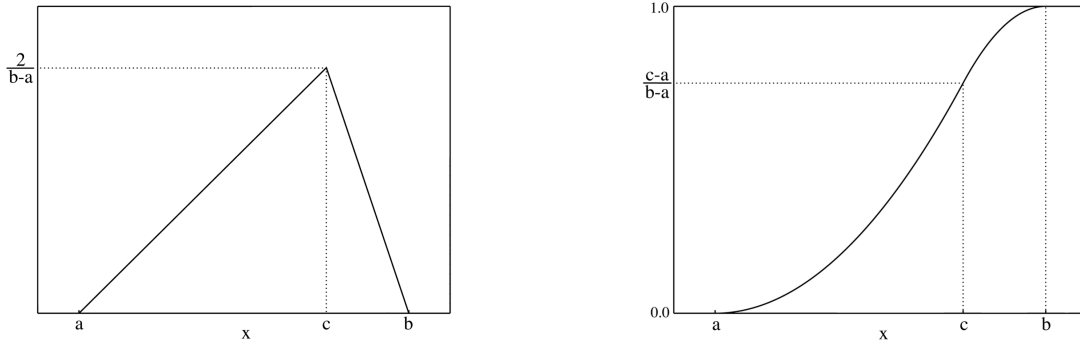


Figure B.4: PDF and CDF of the triangular law

*Random delay generation:*

- let  $z$  be a random number uniformly drawn in  $[0,1]$ .
- let  $F_c = \frac{(c-a)}{(b-a)}$ ,  $\phi_1 = \sqrt{(c-a)(b-a)}$ ,  $\phi_2 = \sqrt{(b-c)(b-a)}$ .
- return:
  - $a$  if  $z = 0$  then return
  - $a + \phi_1 \sqrt{z}$  if  $z < F_c$
  - $b + \phi_2 \sqrt{1-z}$  if  $z < 1$
  - $b$  if  $z = 1$

## B.12 Uniform law

This law is used for delays that are uniformly distributed in a range  $[a, b]$ .

*Syntax:* `uniform(a, b)`

where  $a$  and  $b$  are valid parameters (Section 6.4.1 [Parameters], page 57).

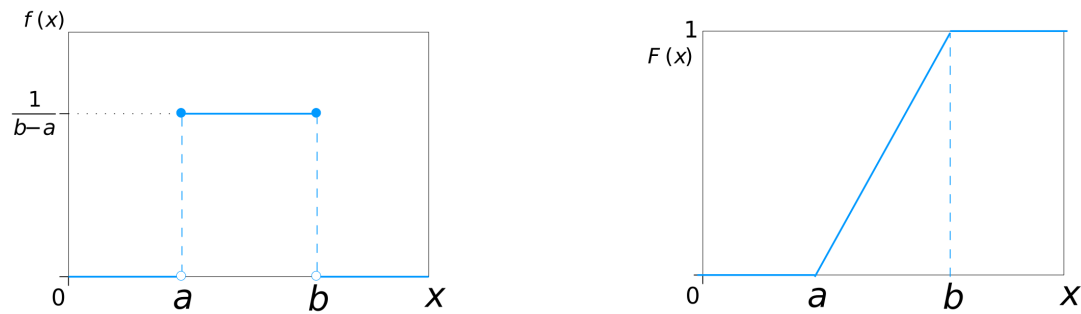


Figure B.5: PDF and CDF of uniform law with parameters  $a$  and  $b$ .

*Random delay generation:*

- let  $z$  be a random number uniformly drawn in  $[0,1]$ .
- return  $a + (b - a)z$

## B.13 Weibull's law

This laws permits to specify Weibull distribution parameterized by its mean  $m$  and its shape  $k$ . Sometimes the scale parameter  $\lambda = \Gamma(1 + \frac{1}{k})/m$  is used instead of  $m$ .

*Syntax:* `weibull(m, k)`

where  $m$  and  $k$  are valid parameters (Section 6.4.1 [Parameters], page 57).

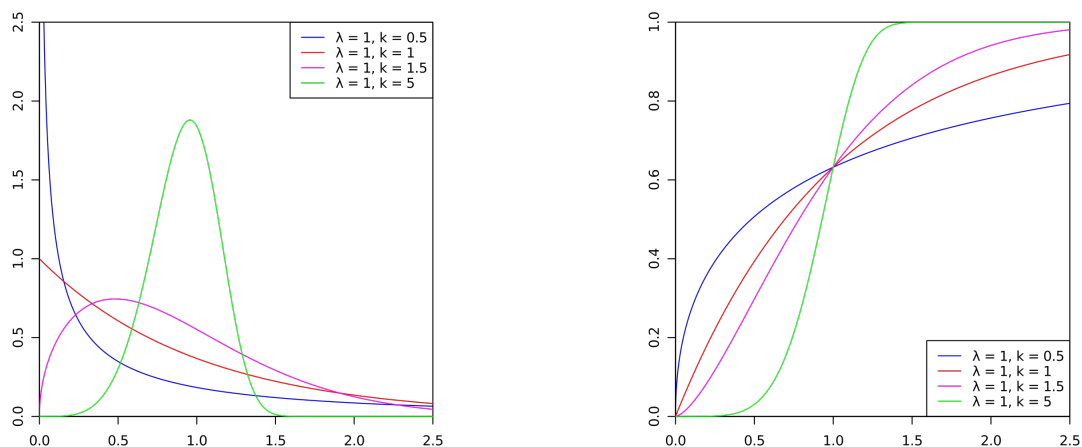


Figure B.6: PDF and CDF of weibull law for different values of the shape  $k$  and scale  $\lambda$ .

*Random delay generation:*

- let  $z$  be a random number uniformly drawn in  $[0,1]$ .
- let  $\lambda = \Gamma(1 + \frac{1}{k})/m$
- return  $(-\ln(z)) \sup 1/k/\lambda$



## B.14 Truncated Weibull's law

This law is the shifted left truncated Weibull distribution [MW89], page 69 described using its mean  $m$ , its shape  $\beta$  and its age (or shift)  $\alpha$ . Its CDF is given by the following formula:

$$F(t) = 1 - \exp(\lambda(\alpha \sup \beta - (t + \alpha) \sup \beta))$$

where  $\lambda = (\frac{\Gamma(1+1/\beta)}{m}) \sup \beta$

*Syntax:* `truncated_weibull(m,  $\beta$ ,  $\alpha$ )`

where  $m$ ,  $\beta$  and  $\alpha$  are valid parameters (Section 6.4.1 [Parameters], page 57).

*Random delay generation:*

- let  $z$  be a random number uniformly drawn in  $[0,1]$ .
- let  $d_0 = \frac{1}{\beta}$
- let  $d_1 = (\frac{\alpha \Gamma(1+d_0)}{m}) \sup \beta$
- return  $\alpha \cdot ((\frac{1-\ln(z)}{d_1}) \sup d_0 - 1)$