NAME

svl, SVL – script language for verification scenarios

DESCRIPTION

SVL (*Script Verification Language*) offers a way to describe (compositional and on-the-fly) verification scenarios, under the form of sequences of statements of several kinds:

- assignment statements, which produce a file containing a representation of a given expression;
- comparison statements, which compare two expressions modulo some equivalence or preorder relation;
- statements for verifying a temporal logic formula in an expression;
- statements for checking livelocks or deadlocks in an expression.

Expressions are built upon the following components:

- behaviour systems represented either implicitly (as LNT, LOTOS, or FSP programs, processes in LNT, LOTOS, or FSP programs, or networks of communicating automata in the EXP format), or explicitly (as Labelled Transition Systems in one of the formats provided with CADP, namely BCG, AUT, sequential FC2, or SEQ); note that the parallel FC2 format is no longer supported;
- LNT and LOTOS like parallel operators, enabling parallel composition of expressions;
- label hiding, label cutting, and label renaming operators;
- operators of abstraction of an expression with respect to some interface;
- operators for generating the explicit LTS of an expression;
- operators of (total or partial) reduction modulo some equivalence relation;
- so-called *meta-operators* of (total or partial) reduction, that apply to SVL abstract trees during a preliminary compilation phase called "expansion"; the meta-operators implement several useful compositional reduction strategies.

Moreover, SVL offers a way to invoke Bourne shell commands and to parameterize expressions with respect to particular tools and methods of CADP.

SYNTAX DESCRIPTION

The syntax of SVL is described using productions in the Extended Bachus-Naur Form (EBNF).

Terminal symbols are written between double quotes, except for the double quote itself, which is written between single quotes. The newline character is written "\n". Non-terminal symbols are written in italic. Optional parts of productions are enclosed between square brackets, and parts that can be iterated zero or more times are written between braces.

The following table sums up the non-terminal symbols and their meaning.

Symbol	Description
P	program, sequence of statements
S, *S1*, *S2*	statement
F	file
EXT	file extension
B	behaviour expression
SPEC	behaviour specification
E	equivalence relation
M	verification method
T	verification tool
RE	result and/or expected value
LL	list of labels
L	label
GPL	list of gate parameters
G	gate
AL	list of data parameters (process)
A	data parameter (process)
TL	list of (possibly typed) labels
LD	list of synchronization definitions
RL	list of renaming rules
LP	label pattern
PID	process/property identifier
C	LNT channel identifier
O	data operator

SYNTAX OF PROGRAMS

A program is a sequence of statements, separated in two categories: *S1* and *S2*. Statements in *S2* are composed sequentially with subsequent statements using the ";" character. Statements in *S1* have their own terminating character and do not need a ";" separator.

P ::= <*empty*>
| *S1* [*P*]
| *S2* [";" *P*]

Statements are either assignments, behaviour comparisons, temporal logic verifications, deadlock/livelock checks, Bourne shell commands, property definitions, or property checks. A property definition can embed other statements. It is given a name, optional parameters, comments, and possibly an expected result that must be attached to each embedded verification statement, namely behaviour comparisons, temporal logic verifications, and deadlock/livelock checks. An expected result may also be attached optionally to Bourne shell commands.

Comparisons can be parameterized modulo a particular equivalence relation (*E*), with a particular CADP tool (*T*), and using a particular exploration method provided by the selected tool (*M*).

Temporal logic formulas can be either stored in a file or inlined in the SVL script.

Deadlock/livelock checks can be parameterized with a particular CADP tool.

Comparisons, temporal logic verifications, and deadlock/livelock checks may produce diagnostic files.

Assignments produce LTS files.

Using the keyword "result *result-id*", one can associate to a comparison, temporal logic verification, deadlock/livelock verification, or Bourne shell command a shell variable *result-id* in which the result (of the verification or Bourne shell command) will be stored. Such shell-variables can be used to guide the script execution depending on verification results, using Bourne shell commands.

```
S1 ::= "%" shell-line "\n"

  | "property" PID ["(" param "," ... "," param ")"]
      ['"' comment '"' ... '"' comment '"']
    "is"
      P
    "end property"

  | [F "="] B "|=" ["using" M] ["with" T] formula ";"

S2 ::= "%" shell-line "\n"
       RE

  | F "=" B

  | "check" PID "(" arg "," ... "," arg ")"

  | [F "="] [E]  ["probabilistic" | "stochastic"]
      "comparison" ["using" M] ["with" T]
      B ("==" | ">=" | "<=") B [ ";" RE ]

  | [F "="] "verify" F ["using" M] ["with" T] "in"
      B [ ";" RE ]

  | [F "="] "deadlock" ["with" T] "of"
      B [ ";" RE ]

  | [F "="] "livelock" ["with" T] "of"
      B [ ";" RE ]

  | [F "="] B "|=" ["using" M] ["with" T] formula ";"
      RE

RE ::= "result" L1
  | "expected" L2
  | "result" L1 "expected" L2

E  ::= "strong" | "observational" | "branching"
  | "divbranching" | "tau*.a" | "safety" | "trace"
  | "weak trace" | "tau-confluence"
  | "tau-compression" | "tau-divergence"

M ::= "std" | "bdd" | "fly" | "bfs" | "dfs"
  | "acyclic"

T  ::= "aldebaran" | "bcg_min" | "bcg_cmp"
  | "bisimulator" | "evaluator" | "evaluator3"
  | "evaluator4" | "evaluator5" | "exhibitor"
```

| "reductor"

Note that the keyword "expected" is only allowed in the scope of a "property" statement.

Note that **fc2tools** are no longer supported.

Files are written between double quotes, and must specify a valid extension. Not all extensions are valid in all contexts, as stated more precisely in the sections describing the particular statements and behaviours.

F ::= '"' *prefix.EXT* '"' | '"' *filename* '"'

EXT ::= "aut" | "bcg" | "fc2" | "seq"
| "lnt" | "lotos" | "lot" | "lts" | "exp"
| "hide" | "hid" | "cut" | "rename" | "ren" | "sync"
| "mcl" | "xtl"

A file *prefix* is any string satisfying the syntax of file names in the current operating system.

SYNTAX OF BEHAVIOUR EXPRESSIONS

Behaviour expressions are built from elementary systems, that are combined together using the various operators described below.

B ::= *SPEC*

| "stop"

| "generation" "of" *B*

| ["leaf" | "root" | "root leaf" | "node" | "smart"]
["total" | "partial"]
[*E*] ["probabilistic" | "stochastic"]
"reduction" ["using" *M*] ["with" *T*]
"of" *B*

| ["total" | "partial" | "gate"] "hide"
(["all" "but"] [*TL*] | "using" *F*) "in" *B*
["end" "hide"]

| ["total" | "partial" | "gate"] "cut"
(["all" "but"] [*TL*] | "using" *F*) "in" *B*
["end" "cut"]

| ["total" | "partial" | "gate"] "prio"
(["all" "but"] *LL* (">" ["all" "but"] *LL*)+)+
"in" *B*
["end" "prio"]

| ["total" | "single" | "multiple" | "gate"]
"rename" (*RL* | "using" *F*) "in" *B*
["end" "rename"]

| ["total" | "partial" | "gate"]
["user"] "abstraction" *B*
["sync" ([*LL*] | "using" *F*)] "of" *B*

```
    |  "refined" ["user"] "abstraction" LL
       ["using" B] "of" B

    | "chaos" ( "with" LL
            | "with" n "labels" LP
            | "using" F )

    | "bag" m ( "with" LL
            | "with" n "labels" LP "," LP
            | "using" F )

    | "fifo" m ( "with" LL
             | "with" n "labels" LP "," LP
             | "using" F )

    | ["label" | "gate"] "par" [("all" | LD) "in"]
      [LD "->"] B ("||" [LD "->"] B)+
      "end par"

    |  B "||" B

    |  B "|||" B

    |  B "|[" [LL] "]|" B

    |  B "-||"["?"] B

    |  B "-|||"["?"] B

    |  B "-|[" [LL] "]|"["?"] B

    |  "(" B ")"

SPEC ::= F
     | [F ":"] L ["[" GPL "]"] ["(" AL ")"]

LL ::= L [ "," LL ]
   |  "{"s t r i n g"}"

GPL ::= L [ "," GPL ]
   |  L "=>" L [ "," GPL ]
   |  "..."
   |  "{"s t r i n g"}"

AL ::= A [ "," AL ]

TL ::= L [ ":" C ] [ "," TL ]
   |  "{"s t r i n g"}"

LD ::= L [ "," LD ]
   |  L "#" n [ "," LD ]
   |  "{"s t r i n g"}"
```

RL ::= *L* "->" *L* ["," *RL*]
| "{" *string* "}"

L ::= *G* | '"' *string* '"'

LP ::= '"' *string-with-%d* '"'

G, *PID*, *C* ::= *lotos-identifier*
| *lnt-identifier*

A ::= *O* ["(" *AL* ")"]
| *A O A*
| *A* "of" *L*
| ["+" | "-"] *n*
| "'" *char* "'"

O ::= *lotos-identifier*
| *lnt-identifier*
| *special-identifier*

where *n* denotes a natural number greater or equal to 2.

A *lotos-identifier* or an *lnt-identifier* is a word that starts with a letter and contains letters, digits, and underscores, and is not a keyword. Underscore can not be the last character of a *lotos-identifier* or of an *lnt-identifier*.

A *special-identifier* is a word consisting of the characters '#', '&', '*', '+', '-', '.', '/', '>', '=', '<', '@', '\', '^', '~', '{', and '}', that is not a key symbol of SVL. Note that if the sequence starts with character '{', then it must finish with character '}'.

PRECEDENCE AND ASSOCIATIVITY OF OPERATORS

- Parallel composition operators ("||", "|||", and "|[" *LL* "]|") associate to the right. For instance,
  "a.bcg" ||| "b.aut" |[A]| "c.fc2" |[B]| "d.seq"
  reads
  "a.bcg" ||| ("b.aut" |[A]| ("c.fc2" |[B]| "d.seq"))
- On the opposite, infix abstraction operators associate to the left. For instance,
  "a.bcg" -||| "b.aut" -|[A]| "c.fc2" -|[B]| "d.seq"
  reads
  (("a.bcg" -||| "b.aut") -|[A]| "c.fc2") -|[B]| "d.seq"
- Infix abstraction has a higher priority than parallel composition. For instance,
  "a.bcg" -|| "i.bcg" ||| "c.bcg" -|| "i.bcg"
  reads
  ("a.bcg" -|| "i.bcg") ||| ("c.bcg" -|| "i.bcg")
- The lexical scope of "hide", "cut", "prio", "rename", "generation", "reduction", and "abstraction" (prefix operators) extends as far as possible to the right of the expression. For instance,
  hide A in "a.bcg" || "b.bcg"
  is the same as
  hide A in ("a.bcg" || "b.bcg")

Examples:

"a.bcg" || "b.bcg" -|| hide G in "c.bcg" ||| "d.bcg"
reads

```
"a.bcg" || ("b.bcg" -|| hide G in ("c.bcg" ||| "d.bcg"))

par A#2 in "a.bcg" || reduction of "b.bcg" || "c.bcg" end par
```
reads
```
par A#2 in "a.bcg" || reduction of ("b.bcg" || "c.bcg") end par
```

SEMANTICS OF BEHAVIOUR EXPRESSIONS

The semantics of behaviours is defined as follows:

BEHAVIOUR SYSTEMS

SPEC may be the name of a file containing a Labelled Transition System (LTS) in one of the AUT (extension **.aut**), BCG (extension **.bcg**), FC2 (extension **.fc2**), or SEQ (extension **.seq**) file formats. BCG files may define stochastic or probabilistic LTSs as explained in the **bcg_min**(LOCAL) manual page.

SPEC may also be the name of a file containing a network of LTSs in the EXP file format (extension **.exp**). See a description of the .exp format in the **aldebaran**(LOCAL) manual page

At last, *SPEC* may also be the name of an LNT, LOTOS, or FSP file, or an instanciation of a process in an LNT, LOTOS, or FSP file. In the latter case, the syntax is as follows:
[*F* ":"] *L* ["[" *GPL* "]"] ["(" *AL* ")"]
where *F* is the name of the LNT (extension **.lnt**), LOTOS (extension **.lotos** or **.lot**), or FSP (extension **.lts**) file, *L* is a label denoting the name of the invoked process, *GPL* is an optional list of gate parameters of the process, and *AL* is an optional list of arguments representing the data parameters of the process. For an FSP process instanciation, *GPL* and *AL* must remain empty. For a LOTOS process instanciation, *GPL* must contain a list of gates. For an LNT process instanciation, *GPL* may use the dot notation "..." and the named notation **`G1 => G2`**, where **`G1`** is a formal gate parameter of the process, and *G2* is the corresponding actual parameter. See the LNT user manual for details. Note that *L*, *GPL*, and *AL* may contain Bourne Shell variables (see Section USING SHELL VARIABLES IN EXPRESSIONS for details). Those Bourne Shell variables can only occur inside double quotes. As a consequence, the double quotes in *AL* must not be interpreted as the delimiter of a LOTOS or LNT character string. For arguments of type string, the double quotes must be escaped, as in the following example:
```
P [A, B] ("$X" of Nat, "\"this is a string\"")
```

The filename *F* is optional. If it is not mentioned, the process will be searched in the default LNT, LOTOS, or FSP file assigned to the shell variable **DEFAULT_PROCESS_FILE** on a shell line preceding the expression (see Section SHELL LINES).

Note that SVL relies uniquely on extensions to recognize file formats. All files describing the behaviour of a system must therefore have a valid extension.

STOP

"stop" represents a Labelled Transition System which contains a single state and no transitions.

HIDING

["total" | "partial" | "gate"] "hide" ["all" "but"] [*TL*] "in"
  B
["end" "hide"]
and
["total" | "partial" | "gate"] "hide" "using" *F* "in"
  B
["end" "hide"]
will hide the labels found in *B* using the given hiding rules. These rules can be specified either as a list *TL* of (possibly typed) labels (first form), or using an external file *F* (second form).

In the first case, the types (usually channel names defined in an LNT program) are purely ignored. SVL builds a temporary file with extension **.hid**, filled with the given labels. In the second case, the hide file must be provided by the user, with extension **.hide** or **.hid**.

A label can be a gate (possibly followed by experiment offers) or a regular expression denoting a gate (possibly followed by experiment offers). For instance, "G", "G.*", "G !1", "G. !.*" are labels. Among them, only "G" is a gate. A channel name can be any LNT identifier (see <lnt-identifier> in Section BEHAVIOUR EXPRESSIONS above), or any string between double quotes.

Double quotes around a label can be omitted if and only if the label is a gate (therefore, the syntax of the LOTOS hiding operator is accepted as a particular case of the more general SVL hiding operator). However, for compatibility with LOTOS syntax, gates that are not enclosed between quotes are systematically turned to uppercase, unless the **-case** option of **svl**(LOCAL) is used. Note that double quotes are mandatory to avoid syntactic ambiguities when a gate has the same name as a reserved SVL keyword (e.g. "reduction", "all", etc.). They are also mandatory to enable the use of shell variables denoting gates or labels as described in Section USING SHELL VARIABLES IN EXPRESSIONS.

The "all but" keywords modify the semantics of the hiding rules: all the labels, except the labels specified in list *TL*, are hidden in the given behaviour.

The keywords "total", "partial", and "gate" modify the matching mode, that is the way the hiding rules are interpreted, see the **caesar_hide_1**(LOCAL) manual page. If no matching mode is specified, then the default is "gate", which implements the LOTOS hiding operator (possibly extended by the use of regular expressions on gate names).

For every hiding with "gate" matching, SVL checks whether the gates to be hidden have an appropriate syntax and emit a warning if they appear to contain experiment offers (which is a common mistake for novice users). For instance,

hide "G !1"

will trigger a warning message because of the occurence of "!1".

Examples:

```
total hide "G"
```

hides every label equal to "G",

```
gate hide "G"
```

hides every label whose gate is G, e.g., "G !1", "G !2",

```
gate hide ".*G.*"
```

hides every label whose gate contains the character G and

```
partial hide "G"
```

hides every label whose gate or offers contain the character G.

See the **bcg_labels**(LOCAL) and **caesar_hide_1**(LOCAL) man pages for more information on the hide file format, and on the semantics of the different matching modes. See the **regexp**(LOCAL) man page for information about the syntax of regular expression.

CUTTING

["total" | "partial" | "gate"] "cut" ["all" "but"] [*TL*] "in"
B
["end" "cut"]
and
["total" | "partial" | "gate"] "cut" "using" *F* "in"
B
["end" "cut"]
will cut the labels found in *B* using the given cutting rules. These rules can be specified either as a list of (possibly typed) labels (first form), or using an external file *F* (second form).

In the first case, the types (usually channel names defined in an LNT program) are purely ignored. SVL builds a temporary file with extension **.cut**, filled with the given labels. In the second case, the cut file must be provided by the user, with extension **.cut**.

A label can be a gate (possibly followed by experiment offers) or a regular expression denoting a gate (possibly followed by experiment offers). For instance, "G", "G.*", "G !1", "G. !.*" are labels. Among them, only "G" is a gate. A channel name can be any LNT identifier (see <lnt-identifier> in Section BEHAVIOUR EXPRESSIONS above), or any string between double quotes.

Double quotes around a label can be omitted if and only if the label is a gate. However, for compatibility with LOTOS syntax, gates that are not enclosed between quotes are systematically turned to uppercase, unless the **-case** option of **svl**(LOCAL) is used. Note that double quotes are mandatory to avoid syntactic ambiguities when a gate has the same name as a reserved SVL keyword (e.g. "reduction", "all", etc.). They are also mandatory to enable the use of shell variables denoting gates or labels as described in Section USING SHELL VARIABLES IN EXPRESSIONS.

The "all but" keywords modify the semantics of the cutting rules: all the labels, except the labels specified in list *TL*, are cut in the given behaviour.

The keywords "total", "partial", and "gate" modify the matching mode, that is the way the cutting rules are interpreted, see the **exp.open**(LOCAL) manual page. If no matching mode is specified, then the default is "gate".

For every cutting with "gate" matching, SVL checks whether the gates to be cut have an appropriate syntax and emit a warning if they appear to contain experiment offers (which is a common mistake for novice users). For instance,

```
cut "G !1"
```

will trigger a warning message because of the occurence of "!1".

Examples:

```
total cut "G"
```

cuts every label equal to "G",

```
gate cut "G"
```

cuts every label whose gate is G, e.g., "G !1", "G !2",

```
gate cut ".*G.*"
```

cuts every label whose gate contains the character G and

```
partial cut "G"
```

cuts every label whose gate or offers contain the character G.

See the **exp.open**(LOCAL) man page for more information on the cut file format, and on the semantics of the different matching modes. See the **regexp**(LOCAL) man page for information about the syntax of regular expression.

PRIORITY

["total" | "partial" | "gate"] "prio"
  (["all" "but"] [*LL*] (">" ["all" "but"] [*LL*])+)+ "in" *B*
["end" "prio"]

sets priorities between the transitions of *B*. In each state of *B*, a transition may be executed only if all transitions of higher priority are not ready for execution.

Priorities between transitions (or equivalently, between labels) are defined by a set of priority rules of the form *X1* > ... > *Xn*, where each *Xi* (for *i* ranging in 1..n) has the form [all but] *LLi* and *LLi* is a list of regular expressions denoting gates or labels. The "all but" keywords that may precede some *LLi* means all gates or labels but those matching *LLi*.

Such priority rules define a transitive relation ">>" on labels as follows:

- if *X* > *X'*, the visible label *L* of *B* matches *X*, and the visible label *L'* of *B* matches *X'* then *L* >> *L'*
- if *L* >> *L'* and *L'* >> *L"* then *L* >> *L"*

L >> *L'* means that any transition labeled *L* has priority over any transition labeled *L'* or, equivalently, any transition labeled *L'* yields priority to any transition labeled *L*.

The relation ">>" must be a strict partial order: *B* must not contain any label *L* such that *L* >> *L*. If ">>" is not a strict partial order, then the **exp.open** tool will issue an error message and then exit.

Beware that the rules *X* > *X'* and *X'* > *X"* (which are equivalent to *X* > *X'* > *X"*) imply *X* > *X"* only if some label of *B* matches *X'*. Therefore, to avoid tricky errors, **exp.open** checks that every individual regular expression *L* in *LL1*, ..., *LLn* matches some label of *B*. If not, then **exp.open** will issue a warning.

The optional "gate", "total", and "partial" keywords define the matching mode, in the same way as for the "hide" and "cut" operators. The matching mode by default is "gate".

See the **exp.open**(LOCAL) manual page for details.

Examples:

```
gate prio
          "A.*" > B > all but "A.*", B
     in
          "f.bcg"
     end prio
```

defines an LTS in which every transition whose gate starts with the letter "A" has priority over every transition whose gate is "B", which themselves have priority over all other transitions.

```
partial prio
          "A" > all but "A"
     in
          "f.bcg"
     end prio
```

defines an LTS in which every transition whose label contains the letter "A" has priority over every transition whose label does not contain the letter "A" (including hidden transitions).

```
total prio
          "A" > "B" > "C"
          "D" > "E" > "F"
          "A" > "D"
          "B" > "E"
          "C" > "F"
     in
          "f.bcg"
     end prio
```

defines an LTS in which:

- A has priority over B, C, D, E, and F,
- B has priority over C, E, and F,
- C has priority over F,
- D has priority over E and F, and
- E has priority over F.

Note: Strong bisimulation is a congruence for all **svl** hiding, cutting, renaming, priority, and parallel composition operators operators. However, branching, divbranching, observational, and safety equivalences are congruences for all **svl** hiding, cutting, renaming, and parallel composition operators, but not for priority. It should also be noted that tau*.a equivalence is not a congruence for parallel composition.

RENAMING

["total" | "single" | "multiple" | "gate"] "rename" *RL* "in"
B
["end" "rename"]
and
["total" | "single" | "multiple" | "gate"] "rename"
"using" *F* "in" *B*
["end" "rename"]
will rename the labels of *B* using the given renaming rules. These rules can be specified either as a list of rules of the form *L1* "->" *L2*, where *L1, L2* denote any labels (first form), or using an external file *F* (second form).

In the first case SVL builds a temporary file with extension **.ren**, filled with the given substitution rules. In the second case SVL uses the given renaming file, whose extension must be **.ren** or **.rename**.

A label can be a gate (possibly followed by experiment offers) or a regular expression denoting a gate (possibly followed by experiment offers). For instance, "G", "G.*", "G !1", "G. !.*" are labels. Among them, only "G" is a gate.

Double quotes around a label can be omitted if and only if the label is a standard LOTOS gate. However, for compatibility with LOTOS syntax, gates which are not enclosed between quotes are systematically turned to uppercase, unless the **-case** option of **svl**(local) is used. Note that double quotes are mandatory to avoid syntactic ambiguities when a gate has the same name as a reserved SVL keyword (e.g. "reduction", "all", etc.) and that they enable the use of shell variables denoting gates or labels as described in Section USING SHELL VARIABLES IN EXPRESSIONS.

The keywords "total", "single", "multiple", and "gate" modify the way the left hand sides of the renaming rules are interpreted, see the **caesar_rename_1**(LOCAL) manual page. If no matching mode is specified, then the default is "gate".

For every renaming with "gate" matching, SVL checks whether the gates to be renamed have an appropriate syntax and emit a warning if they appear to contain experiment offers (which is a common mistake for novice users). For instance,

```
rename "G !1" -> "G !2"
```

will trigger a warning message because of the occurence of "!1". Note however that

```
rename "G" -> "G !1"
```

is correct.

Examples:

```
total rename "G" -> "H"
```

renames to "H" every label equal to "G",

```
gate rename "G" -> "H"
```

renames to "H" the gate of every label whose gate is G, e.g., "G !1", "G !2",

```
gate rename ".*G.*" -> "H"
```

renames to "H" the gate of every label whose gate contains a G,

```
single rename "G" -> "H"
```

replaces the first occurrence of "G" by "H" in every label whose gate or offers contain a G,

```
multiple rename "G" -> "H"
```

replaces every occurrence of "G" by "H" in every label whose gate or offers contain a G, and

```
total rename "\([A-Z0-9]*\) \(!.*\)" -> "\1 !1 \2"
```

inserts "!1" between every gate and its first offer.

See the **bcg_labels**(LOCAL) and **caesar_rename_1**(LOCAL) manual pages for more information on these options and on the format of renaming rules. See also the **regexp**(LOCAL) manual page for more information on regular expressions.

ROOT REDUCTION

"root" ["total" | "partial"]
[*E*] ["probabilistic" | "stochastic"] "reduction"
["using" *M*] ["with" *T*] "of" *B*
or more simply
["total" | "partial"]
[*E*] ["probabilistic" | "stochastic"] "reduction"
["using" *M*] ["with" *T*] "of" *B*
will generate the behaviour *B* (totally or partially) reduced modulo the reduction relation *E*, and possibly taking into account the probabilistic or stochastic information present in *B*. The reduction is done with the tool *T* and using the method *M*.

T, *M*, *E*, "total", "partial", "probabilistic", and "stochastic" are optional:

- Partial reduction is an incomplete form of reduction. Compared to total reduction, it generally generates a larger LTS but in a shorter time. If the "total" or "partial" keyword is not specified, then the reduction will be total.
- The "probabilistic" and "stochastic" reductions are only available for strong, branching, and divbranching bisimulations with bcg_min and using the std method. In this case, the behaviour *B* must denote an explicit LTS containing probabilistic or stochastic information. See the **bcg_min**(LOCAL) manual page for more information.
- The default value for *E* is "strong" and can be changed via the shell variable **DEFAULT_REDUCTION_RELATION**.
- The default value for *M* depends on the equivalence relation considered. The shell variable **DEFAULT_REDUCTION_METHOD** can be set in the SVL file to enforce a particular default value.
- The default value for *T* depends on the relation *E* and whether the reduction is partial or total, as summarized in the following table:

relation	total	partial
strong	**bcg_min**	**reductor**
strong stoch.	**bcg_min**	not available
strong prob.	**bcg_min**	not available
tau-divergence	**reductor+bcg_min**	**reductor**
tau-compression	**reductor+bcg_min**	**reductor**
tau-confluence	**reductor+bcg_min**	**reductor**
branching	**bcg_min**	not available
branching stoch.	**bcg_min**	not available
branching prob.	**bcg_min**	not available
divbranching	**bcg_min**	not available
divbr. stoch.	**bcg_min**	not available
divbr. prob.	**bcg_min**	not available
observational	**aldebaran**	not available
tau*.a	**reductor+bcg_min**	**reductor**
safety	**reductor+bcg_min**	**reductor**
trace	**reductor+bcg_min**	**reductor**
weak trace	**reductor+bcg_min**	**reductor**

where "**reductor+bcg_min**" means that the total *E* reduction is done by first applying partial *E* reduction with **reductor**, followed by total strong reduction with **bcg_min**.

The shell variable **DEFAULT_REDUCTION_TOOL** can be set in the SVL file to enforce a particular default value for *T*.

Example:

```
% DEFAULT_REDUCTION_TOOL="reductor"
% DEFAULT_REDUCTION_RELATION="tau*.a"
"a_red.aut" = total reduction of "a.aut"
```

will induce the total reduction modulo tau*.a of a.aut with **reductor**, and store the result in file a_red.aut.

When a combination of total/partial reduction, tool, method, and relation is not available, SVL tries to change (at run-time) some parameters to perform a (total or partial) reduction as close as possible to what appears to be expected, trying to preserve the parameters in the following priority order: stochastic or probabilistic reduction, reduction relation, reduction tool, reduction method, and then total or partial reduction.

Moreover, if the reduction fails (for instance because of memory exhaustion) then SVL tries to achieve it another way. For instance, using another tool, or performing a reduction modulo a stronger equivalence relation to reduce the size of the LTS to reducee before re-attempting the weaker reduction. When all attempts fail, then the verification proceeds with the non reduced behaviour.

Note: In some versions of CADP, the **aldebaran.old** tool, which performs "observational reduction", may be not available. If this is the case, then SVL replaces "observational reduction" by "branching reduction" (performed by the **bcg_min** tool) and issues a warning message.

LEAF REDUCTION

"leaf" ["total" | "partial"]
[*E*] ["probabilistic" | "stochastic"] "reduction"
["using" *M*] ["with" *T*] "of" *B*

is a meta-operator that will (totally or partially) reduce the LTSs generated as components of *B*. As above, *T*, *M*, *E*, the "total" and "partial" keywords and the "probabilistic" and "stochastic" keywords are optional parameters (see **ROOT REDUCTION**). The final result of a "leaf reduction" is not necessarily as small as that obtained with "root", "root leaf", and "node reduction" since only components of *B* are reduced.

The expansion rules of "leaf reduction" are the following:
leaf reduction of *B* = lRed (*B*)

lRed (*B1* |*op*| *B2*) = lRed (*B1*) |*op*| lRed (*B2*)
lRed (Hide (*X*, *B1* |*op*| *B2*)) = Hide (*X*, lRed (*B1* |*op*| *B2*))
lRed (Hide (*X*, *B*)) = Red (Hide (*X*, lRed (*B*)))
(other cases)
lRed (Cut (*X*, *B1* |*op*| *B2*)) = Cut (*X*, lRed (*B1* |*op*| *B2*))
lRed (Cut (*X*, *B*)) = Red (Cut (*X*, lRed (*B*)))
(other cases)
lRed (Prio (*X*, *B1* |*op*| *B2*)) = Prio (*X*, lRed (*B1* |*op*| *B2*))
lRed (Prio (*X*, *B*)) = Red (Prio (*X*, lRed (*B*)))
(other cases)
lRed (Ren (*X*, *B1* |*op*| *B2*)) = Ren (*X*, lRed (*B1*) |*op*| lRed (*B2*))
lRed (Ren (*X*, *B*)) = Red (Ren (*X*, lRed (*B*)))
(other cases)
lRed (Abs (*B1*, *X*, *B2*)) = Red (Abs (*B1*, *X*, lRed(*B2*)))
lRed (Gen (*B*)) = Red (Gen (lRed (B)))
lRed (*B*) = Red (*B*) otherwise

where Red means reduction, Ren means rename, Abs means abstraction or refined abstraction (its first operand is the interface, and its third operand is the body), Gen means generation, *B*, *B1*, *B2* denote any behaviour, *X* denotes either a file or a list of items (labels or renaming rules), and |*op*| denotes any parallel composition operator.

Note that expansion does not propagate meta-operations across "reduction" operations, nor inside the interface part of the "abstraction" operation.

Note also that, at the end of the expansion phase, the obtained abstract tree is cleaned to optimize execution. Therefore some "reduction" operators inserted by the lRed function are then removed by the cleaning function. For instance, any "reduction" operation inserted at the root of the body of an "abstraction" operation will be systematically deleted to avoid the (useless and expensive) generation of the behaviour to be abstracted. See Section CLEANING below.

ROOT LEAF REDUCTION

"root leaf" ["total" | "partial"]
[*E*] ["probabilistic" | "stochastic"] "reduction"
["using" *M*] ["with" *T*] "of" *B*
is a meta-operator, which has the same meaning as "root reduction of leaf reduction of *B*".

NODE REDUCTION

"node" ["total" | "partial"]
[*E*] ["probabilistic" | "stochastic"] "reduction"
["using" *M*] ["with" *T*] "of" *B*
is a meta-operator that will generate *B* in a compositional way. The only difference between "root leaf" and "node reduction" is that "node reduction" performs also reduction at each parallel composition node, and that hide and cut operators are propagated as far as possible inside the behaviour expression. As above, *T*, *M*, *E*, the "total" and "partial" keywords and the "probabilistic" and "stochastic" keywords are optional parameters (see **ROOT REDUCTION**).

The expansion rules of "node reduction" are the following:
node reduction of B = Red (nRed (B))

nRed (*B1* |*op*| *B2*) = Red (nRed (*B1*) |*op*| nRed (*B2*))
nRed (Hide (*X*, *B*)) = Red (Hide (*X*, nRed (*B*)))
nRed (Cut (*X*, *B*)) = Red (Cut (*X*, nRed (*B*)))
nRed (Prio (*X*, *B*)) = Red (Prio (*X*, nRed (*B*)))
nRed (Ren (*X*, *B*)) = Red (Ren (*X*, nRed (*B*)))
nRed (Abs (*B1*, *X*, *B2*)) = Red (Abs (*B1*, *X*, lRed (*B2*)))
nRed (Gen (*B*)) = Red (Gen (lRed (*B*)))
nRed (*B*) = Red (*B*) otherwise
where Red means reduction, Ren means rename, Abs means abstraction or refined abstraction (its first operand is the interface, and its third operand is the body), Gen means generation, *B*, *B1*, *B2* denote any behaviour, *X* denotes either a file or a list of items (labels or renaming rules), and |*op*| denotes any parallel composition operator.

Note that expansion does not propagate meta-operations across "reduction" operations, nor inside the interface part of the "abstraction" operation. The "node reduction" becomes a "leaf reduction" (lRed) once it has passed through an "abstraction", "refined abstraction", or "generation" operator.

Note also that, at the end of the expansion phase, the obtained abstract tree is cleaned to optimize execution. Therefore some "reduction" operators inserted by the nRed function are then removed by the cleaning function. For instance, any "reduction" operation inserted at the root of the body of an "abstraction" operation will be systematically deleted to avoid the (useless and expensive) generation of the behaviour to be abstracted. See Section CLEANING below.

SMART REDUCTION

"smart" ["total" | "partial"]
[*E*] ["probabilistic" | "stochastic"] "reduction"
["using" *M*] ["with" *T*] "of" *B*

is an operator that will generate *B* in a compositional way using a smart heuristic, the aim being to try to avoid generating too large intermediate LTSs.

To do so, *B* is first turned into a network whose LTSs are minimized. Then, SVL executes the following loop until this network contains only a single LTS:

1 Automatically select several LTSs in the network; this step relies on a heuristic metric (computed by exp2c) that depends on an estimate rate of hidden transitions and an estimate rate of interleaved transitions belonging to the product of selected LTSs.

2 Compose the selected LTSs in parallel.

3 Generate the LTS corresponding to the composition obtained in step 2 and minimize it.

4 Replace in the network the LTSs selected in step 1 by the LTS resulting from step 3; continue in step 1.

The LTS corresponding to the obtained network is finally generated, minimized, and returned as the LTS corresponding to *B*.

The maximal number of LTSs that can be selected in step 1 is bounded. By default, the limit is set to 4. This limit can be changed by assigning a different value to the variable *DEFAULT_SMART_LIMIT*.

Note: Unlike other meta-operations, "smart reduction" is not expanded. Indeed, the expansion phase is static, whereas the order in which the LTSs in *B* are composed by "smart reduction" is determined at run time.

GENERATION

"generation" "of" *B*
will force the generation of an explicit LTS representation of *B*. More precisely, if *B* is an LNT, LOTOS, or FSP program, this program will be compiled, and if *B* has an implicit representation as a network of communicating automata in the EXP format, then an explicit representation of *B* will be generated.

There are some cases where the "generation" operation is implicit:

- generation of the body of a "hide", "cut", or "rename" operation when it is a LNT, LOTOS, or FSP file;
- generation of the interface of an "abstraction" operation;
- generation of the body of a "reduction" operation;

Note that, during the expansion phase, meta-operators do not propagate across "generation" operations in the abstract syntax tree.

PARALLEL COMPOSITION

B "||" *B*
and
B "|||" *B*
and
B "|[" [*LL*] "]|" *B*
have the LOTOS semantics of parallel composition: "|||" denotes parallel composition with synchronization on termination only, "||" denotes parallel composition with synchronization on all gates, and at last "|[" [*LL*] "]|" denotes parallel composition with synchronization on termination and on gates specified in the optional list of labels *LL*.

GENERALIZED PARALLEL

["label" | "gate"] "par" [("all" | *LD*) "in"]
  [*LD* "->"] *B* ("||" [*LD* "->"] *B*)+
"end par"

is an extension of the E-LOTOS/LNT generalized parallel composition operator. It denotes the concurrent execution of parallel behaviours following synchronisation rules expressed using:

- the keyword "all" or the list "*LD*" that follows the keyword "par", called *global synchronisation interface*, and
- the lists *LD* that precede the symbols "->", called *local synchronisation interfaces*.

Synchronisation interfaces *LD* are lists of synchronisation elements of the form *L* or *L#N*, where *L* is a gate or a label and *N* is a natural number called *synchronisation degree*.

The semantics of synchronisation rules are defined in the following paragraphs.

- The "gate" or "label" keywords indicate the *synchronisation mode*, which influences the way synchronisation rules apply to transition labels. We say that a transition matches a synchronisation element of the form *L* or *L#N* in the following cases:
  In "gate" synchronisation mode, a transition matches *L* or *L#N* if the gate of the transition label is *L*. Therefore, for every synchronisation element of the form *L* or *L#N* occurring in a synchronisation interface, *L* must be a gate without offers.
  In "label" synchronisation mode, a transition matches *L* or *L#N* if the transition label is *L*. Therefore, for every synchronisation element of the form *L* or *L#N* occurring in a synchronisation interface, *L* must be a full label (i.e., a gate possibly followed by offers).
  Unlike "hide", "cut", and "rename", regular expressions are not allowed in synchronisation elements. If not specified, the synchronisation mode by default is "gate".
- The keyword "all" is a shorthand notation for the global synchronisation set consisting of all synchronisation elements *L* (without degree) such that *L* is the gate (in "gate" synchronisation mode) or the label (in "label" synchronisation mode) of a transition in at least one of the parallel behaviours, except transitions carrying hidden events.

Synchronisation elements have the following meaning:

- A synchronisation element of the form *L* (without degree) occurring in the global synchronisation interface indicates that all parallel behaviours may synchronize all together on transitions that match *L*.
- A synchronisation element of the form *L* (without degree) occurring in a local synchronisation interface indicates that all parallel behaviours that contain *L* in their synchronisation interface may synchronize all together on transitions that match *L*.
- A synchronisation element of the form *L#N* occurring in the global synchronisation interface indicates that *N* behaviours among the parallel behaviours may synchronize together on transitions that match *L*.
- A synchronisation element of the form *L#N* occurring in a local synchronisation interface indicates that *N* behaviours among the parallel behaviours that contain *L#N* in their synchronisation interface may synchronize on transitions that match *L*.
- A transition in a parallel behaviour may execute asynchronously if both the global synchronization interface and the local synchronization interface of that behaviour do not contain any element of the form *L* or *L#N* such that the transition matches *L*.

Note that both the global synchronisation interface and local synchronisation interfaces may contain several synchronisation elements with same label *L* but different synchronisation degrees. In this case, the corresponding synchronisation rules apply nondeterministically.

Following the above meaning of synchronisation elements, it is possible to prevent the execution of particular transitions matching *L* by using synchronisation elements of the form *L#0*, either in the global synchronisation interface (thus preventing execution of transitions matching *L* in all parallel behaviours) or in local synchronisation interfaces (thus preventing execution of transitions matching *L* in those behaviours containing *L#0* in their interface), provided the (global or local) interface does not contain another occurrence of *L* or *L#N* with *N* a strictly positive number.

Transition synchronisation is a generalization of LOTOS rendezvous: synchronisation requires that all transitions have exactly the same label (i.e., gate and possible offers), which is also the label of the resulting transition.

Note that synchronization interfaces can neither contain the hidden gate nor the termination gate. Behaviours always synchronize on labels whose gate is the termination gate and never synchronize on hidden events.

The following syntactic restrictions (checked by **exp.open**(LOCAL))**should**hold:

- If the global synchronisation interface contains a synchronisation element of the form *L#N* with *N* > 0, then the parallel composition must contain at least *N* parallel behaviours.
- If the global synchronisation interface contains a synchronisation element of the form *L#0*, then no synchronisation element of the form *L* or *L#N* with same *L* and *N* > 0 should occur in the global synchronization interface or in any local synchronization interface.
- If a local synchronisation interface contains a synchronisation element of the form *L* (without degree), then at least two local synchronisation interfaces (this one included) should contain the same synchronisation element.
- If a local synchronisation interface contains a synchronisation element of the form *L#N* with *N* > 0, then at least *N* local synchronisation interfaces (this one included) should contain the same synchronisation element.
- If a local synchronisation interface contains a synchronisation element of the form *L#0*, then no synchronisation element of the form *L* or *L#N* with same *L* and *N* > 0 should occur in the global synchronization interface or in the same local synchronization interface.

ABSTRACTION

["total" | "partial" | "gate"] ["user"] "abstraction" *B*
"sync" [*LL*] "of" *B*
and
["total" | "partial" | "gate"] ["user"] "abstraction" *B*
"sync" "using" *F* "of" *B*
and
["total" | "partial" | "gate"] ["user"] "abstraction" *B*
"of" *B*

denote abstraction using an interface, also called semi-composition. It allows to restrict the rightmost behaviour with respect to its environment i.e., an expression called the interface and a synchronization set.

Interfaces can be either ‘‘exact’’ interfaces i.e., parts of the syntactic environment of the sub-expression to be restricted, or ‘‘user-given’’ interfaces i.e., expressions that are supposed to correctly approximate this environment. The second case must be expressed with the "user" keyword that involves the generation of some validation predicates in the produced LTS. These predicates are checked afterwards, when the components obtained by user abstractions are recomposed together. SVL issues a warning message if the check fails.

Similarly to hide, cut, and rename operators, the synchronization set can be given explicitly as a list of labels in the abstraction expression, or in a **.sync** file. See the **projector**(LOCAL) manual page for more information on the sync file format.

A label can be a gate (possibly followed by experiment offers) or a regular expression denoting a gate (possibly followed by experiment offers). For instance, "G", "G.*", "G !1", "G. !.*" are labels. Among them, only "G" is a gate.

Double quotes around a label can be omitted if and only if the label is a gate . However, gates that are not enclosed between quotes are systematically turned to uppercase, unless the **-case** option of **svl**(LOCAL) is used. Note that double quotes are mandatory to avoid syntactic ambiguities when a gate has the same name as a reserved SVL keyword (e.g. "reduction", "all", etc.). They are also mandatory to enable the use of shell variables denoting gates or labels as described in Section USING SHELL VARIABLES IN

EXPRESSIONS.

The "all but" keywords modify the semantics of the synchronization rules: all the labels, except the labels specified in list *LL*, must be used in the synchronization between the given behaviour and its interface.

The keywords "total", "partial", and "gate" modify the matching mode, that is the way the synchronization rules are interpreted, see the **projector**(LOCAL) manual page. If no matching mode is specified, then the default is "gate".

If no synchronization set is given (no "sync" keyword), the synchronization is done as follows:

- In "gate" matching mode, synchronization is done on all gates visible in the interface.
- In "total" or "partial" matching mode, synchronization is done on all labels visible in the interface.

For every abstraction with "gate" matching, SVL checks whether the gates to be synchronized have an appropriate syntax and emit a warning if they appear to contain experiment offers (which is a common mistake for novice users). For instance,

```
abstraction ... sync "G !1"
```

will trigger a warning message because of the occurence of "!1".

Examples:

```
total abstraction ... sync "G"
```

synchronizes every label equal to "G",

```
gate abstraction ... sync "G"
```

synchronizes every label whose gate is G, e.g., "G !1", "G !2",

```
gate abstraction ... sync ".*G.*"
```

synchronizes every label whose gate contains the character G and

```
partial abstraction ... sync "G"
```

synchronizes every label whose gate or offers contain the character G.

See the **projector**(LOCAL) man page for more information on the sync file format, and on the semantics of the different matching modes. See the **regexp**(LOCAL) man page for information about the syntax of regular expression.

Before doing the semi-composition, SVL does the following to optimize verification efficiency:

- It hides in the interface all gates or labels (depending on the matching mode), except those which are in the synchronization list.
- It reduces the interface modulo safety equivalence.

INFIX ABSTRACTION OPERATORS

B1 "-||" ["?"] *B2*
and
B1 "-|||" ["?"] *B2*
and
B1 "-|[" [*LL*] "]|" ["?"] *B2*
are shorthand notations for, respectively,
["user"] "abstraction" *B2* "of" *B1*
and
["user"] "abstraction" *B2* "sync" "of" *B1*
and
["user"] "abstraction" *B2* "sync" [*LL*] "of" *B1*
where the "?" symbol has the same meaning as the "user" keyword.

REFINED ABSTRACTION

"refined" ["user"] "abstraction" *LL* ["using" *B*] "of" *B*
allows to restrict the rightmost behaviour expression (the *body*) with respect to some of its neighbours

(specified in the list *LL*).

The neighbours are those behaviour systems (LTS files, files containing networks of LTSs, LNT, LOTOS, or FSP files, or processes in LNT, LOTOS, or FSP files) that are in the environment of (i.e., composed in parallel with) the body. The identifier of a neighbour is either its process name (without offer parameters) in the case of a process, or its filename (with extension and between quotes) in all other cases. Each label in *LL* must be the identifier of exactly one neighbour (see examples below).

The behaviour expression that follows the "using" keyword, if present, should provide a set of labels, which includes all labels that can be fired by the body. Its states and transitions are simply ignored. These labels allow to compute the possible synchronizations between the body and its environment, without having to generate the LTS corresponding to the body. This expression is required if the body is an LNT, LOTOS, or FSP file or a process in an LNT, LOTOS, or FSP file and can be omitted otherwise. If the body is a parallel composition expression (encoded in an EXP file) or an EXP file, this set is computed automatically using the **exp.open**(LOCAL) tool. If the body is neither an LNT, LOTOS, or FSP file, a process in an LNT, LOTOS, or FSP file, a parallel composition expression, nor an EXP file, then its LTS is generated and serves as label set.

The "user" keyword should be used when the user cannot guarantee that the label set provided by this expression includes all labels that can be fired by the body. In this case, validation predicates will be generated in the resulting LTS, and warning messages will be issued if the validation predicates are not satisfied.

Refined abstraction executes in two steps: in a first step, an interface and a synchronization set will be generated automatically from the behaviours of neighbours and the environment of the expression, using the **exp.open**(LOCAL) tool; during this step, the LTSs of neighbours in *LL* are automatically minimized modulo safety equivalence, so as to generate as small an interface as possible; in a second step, the interface and synchronization set obtained during the first step are used to restrict the body, using the **projector**(LOCAL) tool.

Examples:

The following expressions are examples of correct usage of the "refined abstraction" operator:

```
"a.bcg" = node strong reduction of
  (
    P
  ||
    (
      (refined abstraction P, "r.bcg" using "q.bcg" of Q)
    ||
      hide A in "r.bcg"
    )
  )
"a.bcg" = generation of
  (
    P
  ||
    (
      refined abstraction P of
        (
          Q
        ||
          (refined abstraction P, Q using "r.bcg" of R)
        )
    )
  )
```

The following expression is an example of incorrect usage of the "refined abstraction" operator, where Q has two neighbours whose identifier is P, and no neighbour whose identifier is R:

```
"a.bcg" = generation of
  (
    P
  ||
    (
      (refined abstraction P, R using "q.bcg" of Q)
    ||
      hide A in P
    )
  )
```

CHAOS AUTOMATA

"chaos" "using" *F*
and
"chaos" "with" *LL*
and
"chaos" "with" *n* "labels" *LP*
generate an LTS with a single state and looping transitions, using the bcg_graph tool.

There are several ways to define the transition labels:

- Using a file *F* (first form): *F* must be the full name of an existing file, no particular extension being required. Each label in *F* must be written on a separate line, and may be enclosed in double quotes that are removed in the generated LTS.
- Using a label list *LL* (second form): *LL* is a list of any length. In this case, SVL generates a temporary file used by bcg_graph.
- Using a label pattern *LP* and a number *n* (third form): *LP* must be enclosed between double quotes and contain exactly one occurrence of the substring "%d". This pattern denotes exactly *n* different labels, obtained by replacing the occurrence of "%d" by numbers in the range 1..*n*. *n* must be a natural number (sequence of digits), possibly enclosed between double quotes. It can also be defined using a shell variable (in which case double quotes are mandatory), such as e.g., "$N" and "${N}0".

See the **bcg_graph**(LOCAL) manual page for more information.

Example:

If "labels" is a file containing three labels A1, A2, and A3, then the following three behaviours denote the same explicit LTS that contains a unique state and three transitions from and to this state, labelled respectively A1, A2, and A3.

```
chaos using "labels"
  chaos with 3 labels "A%d"
  chaos with "A1", "A2", "A3"
```

BAGS AND FIFO BUFFERS

"bag" *m* "using" *F*
and
"bag" *m* "with" *LL*
and
"bag" *m* "with" *n* "labels" *LP1* "," *LP2*
and
"fifo" *m* "using" *F*

and
"fifo" *m* "with" *LL*
and
"fifo" *m* "with" *n* "labels" *LP1* "," *LP2*
generate an LTS modeling a communication buffer of size *m*, which can be either a bag (i.e., a communication buffer in which the ordering of messages is not enforced) or a FIFO (First In/First Out) buffer.

m must be a natural number (sequence of digits), possibly enclosed between double quotes. it can also be defined using shell variables (in which case double quotes are mandatory), such as e.g., "$N" and "${N}0".

Buffers distinguish between two kinds of labels, namely *inputs* and *outputs*, modeling respectively the ingoing and outgoing messages. Each *input* is paired with the corresponding *output*.

There are several ways to define the labels handled by the buffer:

- Using a file *F* (first and fourth forms): *F* must be the full name of an existing file, no particular extension being required. Each label in file *F* must be written on a separate line, and may be enclosed in double quotes that are removed in the generated LTS. Labels occurring in odd (respectively even) positions in the file denote *inputs* (respectively *outputs*). Each *input* is paired with the following *output* label in the file.
- Using a label list *LL* (second and fifth forms): *LL* is a list of even length. Labels occurring in odd (respectively even) positions in the list denote *inputs* (respectively *outputs*). Each *input* is paired with the following *output* in the label list. In this case, SVL generates a temporary label file used by bcg_graph.
- Using two label patterns *LP1*, *LP2* and a number *n* (third and sixth forms): Each of *LP1* and *LP2* must be enclosed between double quotes and contain exactly one occurrence of the substring "%d". These patterns denote exactly 2*n* different labels, obtained by replacing the occurrences of "%d" by numbers in the range 1..*n*. *LP1* (respectively *LP2*) defines the form of *inputs* (respectively *outputs*). *inputs* and *outputs* are paired when they are obtained by instanciating *LP1* and *LP2* with the same number. *n* must be a natural number (sequence of digits), possibly enclosed between double quotes. It can also be defined using shell variables (in which case double quotes are mandatory), such as e.g., "$N" and "${N}0".

See the **bcg_graph**(LOCAL) manual page for more information.

Example:

If "labels" is a file containing four labels INPUT1, INPUT2, OUTPUT1, and OUTPUT2, then the following three behaviours denote the same explicit LTS that models a FIFO buffer with 4 places and exchanging 2 different messages.

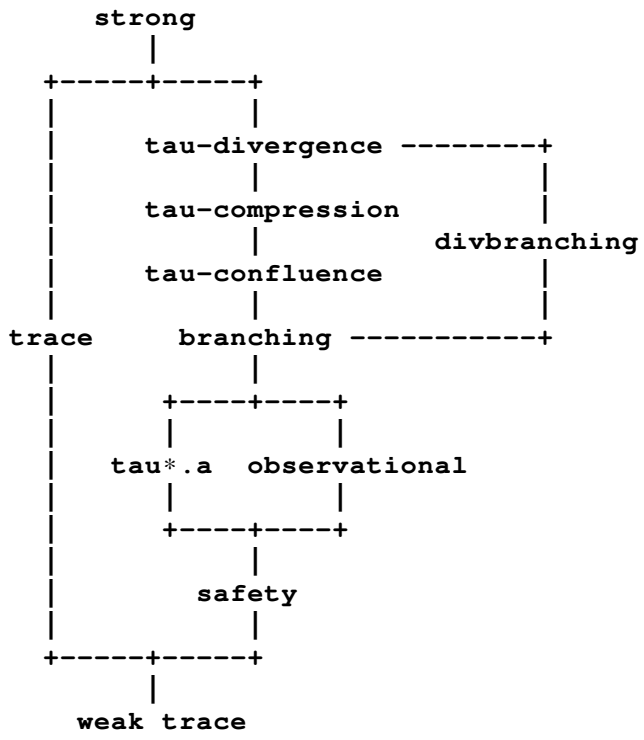
```
fifo 4 using "labels"
   fifo 4 with 2 labels "INPUT%d", "OUTPUT%d"
   fifo 4 with "INPUT1", "OUTPUT1", "INPUT2", "OUTPUT2"
```

CLEANING

Since the expansion phase may generate some redundant operations, the tree of the behaviour is always cleaned using the following clean function:

clean (Gen (Gen (*B*))) = clean (Gen(*B*))
clean (Gen (*SPEC*)) =
    Gen (*SPEC*) if LNT, LOTOS, FSP, or EXP
    SPEC otherwise
clean (Gen (Stop)) = Stop
clean (Gen (Abs (*B1*, *L*, *B2*))) = clean (Abs(*B1*, *L*, *B2*))
clean (Gen (Red (*B*))) = clean (Red (*B*))
clean (Gen (*B*)) = Gen (clean (*B*))
clean (Hide (*L*, Hide (*L'*, *B*)) = clean (Hide (*L* u *L'*, *B*))
        if matching modes are the same
clean (Hide (*X*, Stop)) = Stop
clean (Hide (*X*, *B*)) = Hide (*X*, clean (*B*))
clean (Cut (*L*, Cut (*L'*, *B*)) = clean (Cut (*L* u *L'*, *B*))
        if matching modes are the same
Clean (Cut (*L*, Stop)) = Stop
clean (Cut (*X*, *B*)) = Cut (*X*, clean (*B*))
clean (Prio (*X*, *B*)) = Prio (*X*, clean (*B*))
clean (*B1* |*op*| *B2*) = clean (*B1*) |*op*| clean (*B2*)
clean (Abs (Stop, *L*, *B2*)) = Stop
clean (Abs (*B1*, *L*, *B2*)) = Abs (clean (*B1*), *L*, clean (*B2*))
clean (Red (Red' (*B*))) = clean (Red' (*B*)) if Red' <= Red
clean (Red (Gen (Red (*B*)))) = clean (Red (Gen (*B*)))
clean (Red (Abs (*B1*, *L*, Red (*B2*)))) =
            clean (Red (Abs (*B1*, *L*, *B2*)))
clean (Red (Hide (*L*, Red (*B*)))) = clean (Red (Hide (*L*, *B*)))
clean (Red (Cut (*L*, Red (*B*)))) = clean (Red (Cut (*L*, *B*)))
clean (Red (Ren (*X*, Red (*B*)))) = clean (Red (Ren (*X*, *B*)))
clean (Red (Stop)) = Stop
clean (Red (*B*)) = Red (clean (*B*))
clean (Ren (*X*, Stop)) = Stop
clean (Ren (*X*, *B*)) = Ren (*X*, clean (*B*))
clean (*B*) = *B* otherwise
where Red <= Red' means Red denotes reduction modulo a weaker (<) or equal (=) reduction relation than this of Red'. The "weaker" (partial order) relation is represented by the diagram below, where an arrow

goes from R to R' if R' < R:



SEMANTICS OF STATEMENTS

The semantics of statements is defined as follows:

ASSIGNMENT

F "=" *B*

will store in file *F* the system resulting from the behaviour expression *B*, with possibly format conversion. *F* is created in the current directory.

The file format of *F* can be LOTOS (extension **.lotos** or **.lot**), BCG (extension **.bcg**), AUT (extension **.aut**), SEQ (extension **.seq**), FC2 (extension **.fc2**), or EXP (extension **.exp**). The following rules apply:

- If the file format of *F* is LOTOS, then *B* must be an LNT file (extension **.lnt**). In this case, the LOTOS file *F* is the result of translating the LNT file *B* into LOTOS using **lnt.open**. Note that *F* can only be overwritten by a subsequent assignment statement having explicitly *F* as left-hand side. Therefore, an error message will be issued and the script execution will be stopped if an attempt is made at overwriting *F* in another way, i.e., if an LNT file with same prefix as *F* is used in subsequent behaviour expressions.
- If *B* is an LNT, LOTOS, or FSP file, or if *B* is a network of communicating automata (parallel composition expression) and the format of *F* is this of an explicit LTS, then SVL issues a warning message since the behaviour *B* should be generated before being assigned. However, the generation is done automatically by SVL.

Assignment to EXP files must be used cautiously, as shows the following example.

Consider the following program, where **<B1>**, **<B2>**, and **<B3>** are arbitrary behaviours.

```
"a.bcg" = <B1>;
"c.bcg" = <B2>;
"b.exp" = "a.bcg" ||| "c.bcg";
"a.bcg" = <B3>;
```

```
"d.bcg" = generation of "b.exp"
```

It must be clear that the automaton described in "d.bcg" represents the behaviour **<B3> ||| <B2>** instead of **<B1> ||| <B2>** since at the time "b.exp" is evaluated, "a.bcg" is bound to **<B3>**.

On the contrary, in the following program,

```
"a.bcg" = <B1>;
"c.bcg" = <B2>;
"b.exp" = (reduction of "a.bcg") ||| "c.bcg";
"a.bcg" = <B3>;
"d.bcg" = generation of "b.exp"
```

the automaton described in "d.bcg" represents the behaviour **(reduction of <B1>) ||| <B2>**, since the reduction is evaluated at the time "b.exp" is created.

COMPARISON

[*F* "="] [*E*] ["probabilistic" | "stochastic"] "comparison"
["using" *M*] ["with" *T*]
B ("==" | "<=" | ">=") *B*

allows to compare two behaviours. Symbol "==" means "equivalence" whereas "<=" and "=>" denote relation pre-orders. The optional file *F* must have extension **.aut**, **.bcg**, **.fc2**, or **.seq**. It is created in the current directory and may contain a diagnostic of the comparison if the result is FALSE.

Some combinations of tool, method, and relation are not available. In this case, SVL tries to change some parameters to perform a comparison as close as possible to what seems to be expected. As much as possible, SVL tries to preserve the parameters in the following priority order: relation, tool, and then method.

T, *M*, *E*, "probabilistic", and "stochastic" are optional:

- The "probabilistic" and "stochastic" comparisons are only available for strong, branching, and divbranching bisimulations with bcg_cmp and using the std method. In this case, the behaviours *B* must denote explicit LTSs containing probabilistic or stochastic information. See the **bcg_cmp**(LOCAL) manual page for more information.
- The default value for *E* is "strong" and can be changed via the shell variable **DEFAULT_COMPARISON_RELATION**.
- The default value for *M* depends on the equivalence relation considered. The shell variable **DEFAULT_COMPARISON_METHOD** can be set in the SVL file to enforce a particular default value.
- The default value for *T* depends on the type of comparison (pre-order or equivalence), on the relation *E*, and on whether at least one behaviour *B* is an implicit LTS or not. For pre-order comparisons, the default value for *T* is *bisimulator*. For equivalence comparisons, the default value for *T* is determined as in the following table:

relation	both explicit	implicit
strong	**bcg_cmp**	**bisimulator**
strong stoch.	**bcg_cmp**	not available
strong prob.	**bcg_cmp**	not available
tau-divergence	not available	
tau-compression	not available	
tau-confluence	not available	
branching	**bcg_cmp**	**bisimulator**
branching stoch.	**bcg_cmp**	not available

branching prob.	**bcg_cmp**	not available
divbranching	**bcg_cmp**	not available
divbr. stoch.	**bcg_cmp**	not available
divbr. prob.	**bcg_cmp**	not available
observational	**bcg_cmp**	not available
tau*.a	**bisimulator**	
safety	**bisimulator**	
trace	**bisimulator**	
weak trace	**bisimulator**	

If a relation is not available for implicit LTSs, then a warning is issued and the LTSs are automatically converted into explicit LTSs.

The shell variable **DEFAULT_COMPARISON_TOOL** can be set in the SVL file to enforce a particular default value for *T*.

VERIFICATION

[*F1* "="] "verify" *F2* ["using" *M*] ["with" *T*] "in" *B*
allows to evaluate a formula on a behaviour, with **evaluator3**(LOCAL), **evaluator4**(LOCAL), **evaluator5**(LOCAL), or **xtl**(LOCAL), and using method *M*. The formula must be written in file *F2*, either in MCL (Model Checking Language) version 3 (regular alternation-free mu-calculus, see the **mcl3**(LOCAL) manual page), in MCL version 4 (value-passing modal mu-calculus, see the **mcl4**(LOCAL) manual page), in MCL version 5 (probabilistic value-passing modal mu-calculus, see the **mcl5**(LOCAL) manual page), or in XTL (eXecutable Temporal Language, see the **xtl-lang**(LOCAL) manual page). If the formula is written in (dataless or full) MCL, then the file *F2* must have extension **.mcl**. If the formula is written in XTL, then the file *F2* must have extension **.xtl**.

The "with *T*" clause is optional. *T* may be one of "evaluator3" (corresponding to the tool **evaluator3**(LOCAL)), "evaluator4" (corresponding to the tool **evaluator4**(LOCAL)), "evaluator5" (corresponding to the tool **evaluator5**(LOCAL)), "evaluator" (corresponding to the tool **evaluator**(LOCAL)),**or** "xtl" (corresponding to the tool **xtl**(LOCAL)). See **evaluator**(LOCAL) for details on the differences between "evaluator", "evaluator3", "evaluator4", and "evaluator5".

If the "with *T*" clause is not present, then:

- If the file *F2* has extension **.xtl**, then the tool is **xtl**.
- If the file *F2* has extension **.mcl**, then the tool is given by the shell variable **DEFAULT_VERIFY_TOOL**, which is set to "evaluator" by default. Note that in this case, **DEFAULT_VERIFY_TOOL** should not have value "xtl". If the file *F2* has extension **.xtl**, then the value of the shell variable **DEFAULT_VERIFY_TOOL** is irrelevant.

The "using *M*" clause is optional, and irrelevant in the case of an XTL formula. In the case of an MCL formula, *M* may be one of "dfs", "bfs", or "acyclic". See the **evaluator**(LOCAL), **evaluator3**(LOCAL), **evaluator4**(LOCAL), or **evaluator5**(LOCAL) manual pages for details about these methods. If the "using *M*" clause is not present, then the method is given by the shell variable **DEFAULT_VERIFY_METHOD**, which is set to "dfs" by default.

File *F1* is optional and must have extension **.aut**, **.bcg**, **.fc2**, or **.seq**. If present, it will contain a diagnostic of the verification in the case of an MCL formula. It is irrelevant in the case of an XTL formula, since the tool **xtl** does not generate counter-examples.

VERIFICATION OF INLINE FORMULAS

[*F* "="] *B* "|=" ["using" *M*] ["with" *T*] *formula* ";"
is similar to "verify" (see Section VERIFICATION for details), except that the formula is not stored in a

file, but inlined in the SVL script.

The formula (defined by symbol *formula* in the syntax above) is defined as the sequence of characters starting from the first character following "|=" (or following the optional "using *M*" and "with *T*", if any, where *M* ranges over "std", "dfs", "bfs", and "acyclic", and *T* ranges over "evaluator", "evaluator3", "evaluator4", "evaluator5", and "xtl") and ending at the last character preceding the next ";", including all spaces and comments. Therefore, ";" must be present at the end of this statement, even if it is the last statement of the SVL script.

MCL or XTL libraries can be included by default in the formulas, by using the variables **DEFAULT_MCL_LIBRARIES** (libraries included by default in all MCL formulas), **DEFAULT_EVALUATOR3_LIBRARIES** (libraries included by default in dataless MCL formulas checked by **evaluator3**(LOCAL)), **DEFAULT_EVALUATOR4_LIBRARIES** (libraries included by default in full MCL formulas checked by **evaluator4**(LOCAL)), **DEFAULT_EVALUATOR5_LIBRARIES** (libraries included by default in full MCL formulas checked by **evaluator5**(LOCAL)), and **DEFAULT_XTL_LIBRARIES** (libraries included by default in all XTL formulas). These variables may be either empty, or contain a list of MCL or XTL library names (with extensions) separated by commas. They are initially empty, meaning that no library is included by default.

The formula (together with its included libraries) is stored in an intermediate file and parsed by the appropriate tool only during script execution. Therefore, syntactic errors in the formula (or in the included libraries) will be detected only at runtime, details being reported in the log file.

If the "with *T*" clause is absent, then the tool is given by the shell variable **DEFAULT_VERIFY_TOOL**, which is set to "evaluator" by default. Therefore, by default the formula is supposed to be a dataless MCL formula.

Note that shell variables are substituted in formulas.

Examples:

```
% ACTION=A
"diag.bcg" = "model.lnt" |= using dfs < '$ACTION' > true;

% DEFAULT_XTL_LIBRARIES="actl.xtl"

"model.bcg" |= with xtl
     let PUT : labelset = EVAL_A (PUT) in
          PRINT_FORM (AG_A (not (PUT), EF (Dia (PUT, true))))
          nop
     end_let;

(* the following is equivalent to the above *)
% DEFAULT_VERIFY_TOOL=xtl
"model.bcg" |=
     let PUT : labelset = EVAL_A (PUT) in
          PRINT_FORM (AG_A (not (PUT), EF (Dia (PUT, true))))
          nop
     end_let;
```

DEADLOCK AND LIVELOCK CHECKING

[*F* "="] "deadlock" ["with" *T*] "of" *B*
and
[*F* "="] "livelock" ["with" *T*] "of" *B*
allow to search deadlocks, respectively livelocks, in a behaviour. They return TRUE if the behaviour has at least one deadlock, respectively likelock, and FALSE otherwise. The optional file *F* must have extension **.aut**, **.bcg**, **.fc2**, or **.seq**. It is created in the current directory and may contain a diagnostic of the search, if

such a diagnostic is available.

The tool *T* is an optional parameter, similar to those described above for the reductions and comparisons. The default value may be modified via the following variables:

- **DEFAULT_DEADLOCK_TOOL**, whose initial value is "exhibitor". This variable can alternatively be set to "aldebaran", "evaluator", "evaluator3", "evaluator4", or "evaluator5".
- **DEFAULT_LIVELOCK_TOOL**, whose initial value is "evaluator". This variable can alternatively be set to "evaluator3", "evaluator4", "evaluator5", or "aldebaran".

Those two shell variables replace the shell variable **DEFAULT_LOCK_TOOL**, which was present in earlier versions of SVL and is now obsolete.

Note that **fc2tools** are no longer supported.

PROPERTY

"property" *PID* ["("*param*"," ..."," *param*")"]
['"'*comment*'"' ... '"'*comment*'"'] "is" *P*
"end property"
is the "property" statement, which allows a set of statements to be attached the following attributes:

- The property has a name, defined by the identifier *PID*. It is a string that starts with a letter and contains letters, digits, and underscores, but does not end with an underscore.
- The property may be parameterized with an arbitrary number of parameters. Each parameter *param* is an identifier satisfying the same syntax as property identifiers. Distinct parameters of the same property must have distinct identifiers. Inside the property, every parameter *param* can be expanded using $*param* or ${*param*} as ordinary shell variables. In particular, if the property contains inline formulas, the parameters are expanded in the formulas.
- The property may have comments consisting of a (possibly empty) sequence of character strings, each enclosed in double quotes. Note that parameters can be used inside comments.

The body *P* of the property consists of a (possibly empty) sequence of statements. Among those statements, *verification statements* are the behaviour comparison statement (see Section COMPARISON), the temporal logic verification statement (see Sections VERIFICATION and VERIFICATION OF INLINE FORMULAS), and the deadlock and livelock checking statements (see Section DEADLOCK AND LIVELOCK CHECKING).

Each verification statement (if any) embedded in the property can be followed by an expected result, in the form "expected *L*", where *L* is either an identifier (e.g., TRUE, FALSE, ...) or an arbitrary string, which may be a Unix regular expression (see the **regexp**(5) manual page for details about regular expressions). A verification statement can be followed by an expected result if and only if all verification statements occurring in the same property are also followed by an expected result. Note that expected results are not allowed outside a property.

Note that for a comparison, a deadlock or livelock checking, or the verification of a formula using **evaluator3**(LOCAL) or **evaluator4**(LOCAL), the result is either TRUE or FALSE. For the verification of a formula using **evaluator5**(LOCAL), the TRUE or FALSE result may be preceded by probability values if requested by the probabilistic operators contained in the formula. The probability values are not considered part of the result, only the TRUE or FALSE result being considered. For the verification of a formula using **xtl**(LOCAL), the format of the result depends on the XTL program; SVL only considers the last non-empty line printed by XTL as being the result. In all cases, we say that the result satisfies the expected result if the expected result matches the result entirely, using the egrep command (see the **egrep**(1) manual page for details).

If a property has no parameter, then it is checked automatically. Otherwise, it must be instanciated using a "check" statement of the following form:
"check" *PID* "(" *arg* "," ... "," *arg* ")"

where each *arg* is an argument, which may be any identifier, string (in double quotes), or natural number (i.e., sequence of digits), and where the number of arguments must be equal to the number of parameters.

When checking a property, SVL will display the following information to the user:

- the name of the property, together with the argument values
- the comments (one line per comment)
- if expected results are defined, a PASS or FAIL verdict; PASS indicates that all verification statements met the expected results; otherwise, the verdict has either the form FAIL if the property contains a single verification statement, or the form "FAIL (*N*/*M*)" such that *M* (where $M > 1$) is the total number of verification statements contained in the property and *N* (where $0 <= N < M$) is the number of verification statements that met the expected results
- if expected results are not defined, the results of verifications in sequence, in the order of statements

More details of the verifications can be displayed if the user sets the shell variable **PROPERTY_DISPLAY_MODE** to a non-zero value.

Examples:

```
property No_Deadlock
    "There should be no deadlock in the specification"
is
    "diag.seq" = deadlock of "spec.bcg";
        expected FALSE
end property (* checked automatically *)

property Exist (A, SPEC)
    "Action \"$A\" should be reachable in the specification"
is
    "diag_$A_$SPEC.seq" =
        "$SPEC.bcg" |= < true* . '$A' > true;
            expected TRUE
end property

check Exist ("G", "my_spec");
check Exist ("H", "my_spec")
```

The display of comments can be parameterized by the user. SVL provides two means to do so:

- Either redefine the three shell variables **PROPERTY_COMMENT_OPEN**, **PROPERTY_COMMENT_MIDDLE**, and **PROPERTY_COMMENT_CLOSE**, which specify the open comment symbol (printed on one separate line if different from the empty string), middle comment symbol (which will start every comment line), and close comment symbol (printed on one separate line if different from the empty string). See examples below.
- Or use one of the predefined comment styles by calling the shell function **SVL_SET_PROPERTY_COMMENT_STYLE** with a parameter among **silent** (comments are not displayed), **none**, **standard** (style by default), **indent**, **ada**, **c**, **pascal**, or **sh**. Styles are illustrated in the following table:

style	open symbol	mid symbol	close symbol
none			
standard		" | "	
indent		" "	
ada		"-- "	
c	"/*"	" * "	" */"
pascal	"(*"	" * "	" *)"
sh		" # "	

Example:

The following SVL code:

```
% PROPERTY_COMMENT_OPEN="(+"
% PROPERTY_COMMENT_MIDDLE=" + "
% PROPERTY_COMMENT_CLOSE=" +)"

property P
    "this is a comment for property P"
is
    stop |= < A > true;
end property
```

will execute as follows:

```
property P
(+
 + this is a comment for property P
 +)

FALSE
```

Example:

The following SVL code:

```
% SVL_SET_PROPERTY_COMMENT_STYLE ada

property P
    "this is a comment for property P"
is
    stop |= < A > true;
end property
```

will execute as follows:

```
property P
-- this is a comment for property P

FALSE
```

SHELL LINES

SVL offers the facility to insert shell lines between statements. Such lines must start with the symbol "%", and finish with the end of the line.

Of course, SVL operates no static control of the validity of inserted shell lines. Hence, erroneous shell scripts may be generated by SVL due to syntax errors made by the user, or to shell lines breaking the consistency of SVL execution.

COMMENTS

SVL accepts two kinds of comments:

- Every sequence of characters delimited by "(*" and "*)" is a comment. This notation is inherited from LOTOS and also available in LNT.
- Every sequence of characters from "--" until the next newline character is a comment. This notation is inherited from LNT.

Comments are ignored (i.e., considered as a blank character) at any place in an SVL script, except:

- within character strings (delimited by double quotes), and
- in shell lines (starting with "%"), where shell comments (starting with "#") should be used instead.

STORING STATEMENT RESULTS IN SHELL VARIABLES

SVL allows the result of some statements to be stored in shell variables using the keyword "result" followed by a label, which denotes a shell expression that expands to a shell variable identifier. The statements whose result can be stored are Bourne shell commands, comparison statements, temporal logic verification statements, and deadlock/livelock checking statements. If a statement (e.g., a Bourne shell command or an XTL property) produces several lines of output, then only the contents of the last non-empty line is stored in the shell variable.

Example 1:

```
"a.bcg" = comparison "spec.bcg" == "serv.bcg";
        result R;

% if [ "$R" = FALSE ]
% then
        -- other verifications
        ...
% fi
```

Example 2:

```
property P1 (R1)
    "a property"
is
    -- equivalence checking
    "a.bcg" = comparison "spec.bcg" == "serv.bcg";
        result "$R1" expected TRUE;

    -- other verifications
    ...
end property

property P2 (R2)
```

```
    "another property reusing part of the previous one"
is
    -- use $R2 to avoid checking equivalence again
    % echo "$R2"
        expected TRUE;

    -- other verifications
    ...
end property

check P1 ("Result"); -- assigns variable Result
check P2 ("$Result") -- uses the value assigned to Result
```

Note that "result" can be used in any context, unlike "expected", which cannot be used outside the context of a "property". Also, if a statement uses both "result" and "expected", then:

- the "result" part must occur before the "expected" part, and
- the "result" and "expected" parts must not be separated by a semicolon.

See the syntax of non-terminal *RE* in Section SYNTAX OF PROGRAMS AND EXPRESSIONS above.

USING SHELL VARIABLES IN EXPRESSIONS

Every string between double quotes may use defined Bourne shell variables ("$"*defined-shell-variable*). This includes the shell variables defined using the "result" keyword (see Section STORING STATEMENT RESULTS IN SHELL VARIABLES above) and the special Bourne shell variables "$#", "$*", "$@", "$1", "$2", ... which can be used to refer to the *script-parameters* passed to the generated script.

However, remember that when using filenames, extensions must always be explicitly mentioned, for the following reasons:

- They permit to determine at compile-time the resulting format of a behaviour;
- They permit to distinguish between a file name and a process name, as illustrated in some of the examples below.

Examples:

```
% for FNAME in a b c
% do
"reduced-$FNAME.aut" = reduction of "$FNAME.aut"
% done
```

is correct, but

```
% for FNAME in a.aut b.aut c.aut
% do
"reduced-$FNAME" = reduction of "$FNAME"
% done
```

is not correct (a run-time error is issued) because "$FNAME" is interpreted at compile-time as a process whereas it denotes a file name. However, the following example is correct:

```
% DEFAULT_PROCESS_FILE="a.lotos"
% for PNAME in "P1[G1, G2]" P2
% do
"reduced-$PNAME" = reduction of generation of "$PNAME
% done
```

This loop reduces in turn the LTSs of processes **P1[G1, G2]** and **P2** found in file **"a.lotos"**.

Shell variables can also be used to denote gates or labels as in the following example:

```
% for G in PUT GET
% do
  "P_$G.exp" =
      "spec.lotos":P ["$G"]
      |["$G"]|
      "spec.lotos":P ["$G"];
% done
```

This loop generates in turn two composition expressions stored in EXP files, corresponding to **"spec.lotos":P [PUT] |[PUT]| "spec.lotos":P [PUT]** and **"spec.lotos":P [GET] |[GET]| "spec.lotos":P [GET]**.

Note also that shell variables and expressions can be used to denote lists of labels, gates, or renaming rules, using the braced notation. A shell expression written between braces will be interpreted as a list of labels instead of a single label, thus differentiating from the quoted notation.

For instance, one may write

```
% L1="A,
"f1.bcg" = total hide D, {$L1} in "f2.bcg";

% L2="A, B, C"
"f3.bcg" = "f4.bcg" |[ {$L2} ]| "f5.lotos":P ["D", {$L2}];

% L3="
% L4="C -> A, B -> C"
"f6.bcg" = total rename "D" -> "E", {$L3, $L4} in "f7.bcg";
```

to express that $A, $B, $C must not be interpreted as single labels or rules (as it would be in "$A", "$B", "$C") but as lists of labels or rules.

Note that the label and rule separator is the coma. However, comas between parentheses are not interpreted as label or rule separators, as long as parentheses are well-balanced.

LOCAL SHELL VARIABLES

SVL provides several shell variables that allow users to fine-tune their verification scripts. Those variables may be modified (carefully) by the user, in a shell line, as already mentionned.

Use of variables whose name starts with **SVL_** should be avoided. This syntax is reserved to functions and variables defined internally by SVL.

DEFAULT VERIFICATION PARAMETERS

The following table lists the user-redefinable shell variables relative to the choice of tools, verification methods, equivalence relations, etc., together with their default values.

Variable	Default	Alternative
DEFAULT_REDUCTION_TOOL	See Sect. REDUCTION above	aldebaran bcg_min reductor
DEFAULT_COMPARISON_TOOL	See Sect. COMPARISON above	aldebaran bcg_cmp bisimulator
DEFAULT_VERIFY_TOOL	See Sect. VERIFY above	evaluator evaluator3 evaluator4 evaluator5 xtl
DEFAULT_DEADLOCK_TOOL	exhibitor	aldebaran evaluator evaluator3 evaluator4 evaluator5
DEFAULT_LIVELOCK_TOOL	evaluator	evaluator3 evaluator4 evaluator5 aldebaran
DEFAULT_REDUCTION_METHOD	std	bdd fly
DEFAULT_COMPARISON_METHOD	dfs	std bdd fly bfs dfs
DEFAULT_VERIFY_METHOD	dfs	bfs acyclic
DEFAULT_REDUCTION_RELATION	strong	observational tau*.a branching divbranching safety tau-compression tau-divergence tau-confluence trace weak trace
DEFAULT_COMPARISON_RELATION	strong	observational tau*.a branching divbranching safety trace weak trace

DEFAULT_PROCESS_FILE	not set	
DEFAULT_SMART_LIMIT	4	any nat > 1
DEFAULT_MCL_LIBRARIES	""	comma-separated MCL lib list
DEFAULT_EVALUATOR3_LIBRARIES	""	comma-separated MCL3 lib list
DEFAULT_EVALUATOR4_LIBRARIES	""	comma-separated MCL4 lib list
DEFAULT_EVALUATOR5_LIBRARIES	""	comma-separated MCL5 lib list
DEFAULT_XTL_LIBRARIES	""	comma-separated XTL lib list

Note: Variable **DEFAULT_LOTOS_FILE** is deprecated. Instead, it is recommended to use **DEFAULT_PROCESS_FILE**, which can be any file containing a LOTOS (extensions **.lot** and **.lotos**), LNT (extension **.lnt**), or FSP (extension **.lts**) program. However, scripts using **DEFAULT_LOTOS_FILE** should continue to work correctly.

TOOL OPTIONS

The following table lists user-redefinable shell variables that may be assigned options that are passed to tools. By default, these variables are empty. See the respective tool manual pages for information about the available options.

Variable	Role
ALDEBARAN_OPTIONS	options passed to **aldebaran**
BCG_CMP_OPTIONS	options passed to **bcg_cmp**
BCG_GRAPH_OPTIONS	options passed to **bcg_graph**
BCG_IO_OPTIONS_INPUT	input options passed to **bcg_io**
BCG_IO_OPTIONS_OUTPUT	output options passed to **bcg_io**
BCG_LABELS_OPTIONS	options passed to **bcg_labels**
BCG_MIN_OPTIONS	options passed to **bcg_min**
BCG_OPEN_OPTIONS	options passed to **bcg_open**
BCG_OPEN_CC_OPTIONS	C compiler options passed by **bcg_open**
CAESAR_ADT_OPTIONS	options passed to **caesar.adt**

CAESAR_OPTIONS	options passed to **caesar**
EVALUATOR_OPTIONS	options passed to **evaluator**
EVALUATOR4_OPTIONS	options passed to **evaluator4**
EVALUATOR5_OPTIONS	options passed to **evaluator5**
EXHIBITOR_OPTIONS	options passed to **exhibitor**
EXP_OPEN_OPTIONS	options passed to **exp.open**
EXP_OPEN_CC_OPTIONS	C compiler options passed by **exp.open**
FSP_OPEN_OPTIONS	options passed to **fsp.open**
FSP_OPEN_CC_OPTIONS	C compiler options passed by **fsp.open**
GENERATOR_OPTIONS	options passed to **generator**
LNT_OPEN_OPTIONS	options passed to **lnt.open**
LNT_OPEN_CC_OPTIONS	C compiler options passed by **lnt.open**
LOTOS_OPEN_OPTIONS	options passed to **lotos.open**
LOTOS_OPEN_CC_OPTIONS	C compiler options passed by **lotos.open**
PROJECTOR_OPTIONS	options passed to **projector**
REDUCTOR_OPTIONS	options passed to **reductor**
SEQ_OPEN_OPTIONS	options passed to **seq.open**
SEQ_OPEN_CC_OPTIONS	C compiler options passed by **seq.open**
XTL_OPTIONS	options passed to **xtl**

As of February 2020, the variables **CAESAR_OPEN_OPTIONS** and **CAESAR_OPEN_CC_OPTIONS** are obsolete and replaced by **LOTOS_OPEN_OPTIONS** and **LOTOS_OPEN_CC_OPTIONS**, respectively.

In general, SVL does not check the contents of these variables before passing them to the corresponding tools. However, there is a notable exception concerning variable **EXP_OPEN_OPTIONS**:

- Before calling **exp.open**, SVL checks that **$EXP_OPEN_OPTIONS** contains at most one option among the partial order reduction options available in **exp.open**, namely **-branching**, **-deadpreserving**, **-strong**, and **-weaktrace**. Otherwise it issues an error message. See **exp.open**(LOCAL)

manual page for more details about partial order reduction options.

- In normal functioning, SVL tries to infer a partial order reduction option from the context of the composition expression in the SVL program, and then calls **exp.open** with this option. However, this is not done if **$EXP_OPEN_OPTIONS** already contains a partial order reduction option, which is thus given priority over the one that could be inferred.

LOCATION OF EXECUTABLES

The following table lists user-redefinable shell variables that may be assigned a path to a different version of the executable file related to a given tool.

Variable	Default value
ALDEBARAN_EXECUTABLE	**aldebaran**
BCG_CMP_EXECUTABLE	**bcg_cmp**
BCG_GRAPH_EXECUTABLE	**bcg_graph**
BCG_IO_EXECUTABLE	**bcg_io**
BCG_LABELS_EXECUTABLE	**bcg_labels**
BCG_MIN_EXECUTABLE	**bcg_min**
BCG_OPEN_EXECUTABLE	**bcg_open**
BISIMULATOR_EXECUTABLE	**bisimulator**
CAESAR_ADT_EXECUTABLE	**caesar.adt**
CAESAR_EXECUTABLE	**caesar**
EVALUATOR_EXECUTABLE	**evaluator**
EVALUATOR3_EXECUTABLE	**evaluator3**
EVALUATOR4_EXECUTABLE	**evaluator4**
EVALUATOR5_EXECUTABLE	**evaluator5**
EXHIBITOR_EXECUTABLE	**exhibitor**
EXP_OPEN_EXECUTABLE	**exp.open**
FSP_OPEN_EXECUTABLE	**fsp.open**
GENERATOR_EXECUTABLE	**generator**
LNT_DEPEND_EXECUTABLE	**lnt_depend**
LNT_OPEN_EXECUTABLE	**lnt.open**

LOTOS_OPEN_EXECUTABLE	**lotos.open**
PROJECTOR_EXECUTABLE	**projector**
REDUCTOR_EXECUTABLE	**reductor**
SEQ_OPEN_EXECUTABLE	**seq.open**
XTL_EXECUTABLE	**xtl**

As of February 2020, the variable **CAESAR_OPEN_EXECUTABLE** is obsolete and replaced by **LOTOS_OPEN_EXECUTABLE**.

PROPERTY DISPLAY PARAMETERS

The following table lists the user-redefinable variables that may be used to parameterize the way messages are printed when evaluating properties. See Section PROPERTY for details.

Variable	Role	Value
PROPERTY_DISPLAY_MODE	displayed information	0/1
PROPERTY_COMMENT_OPEN	comment opening symbol	any
PROPERTY_COMMENT_MIDDLE	comment middle symbol	any
PROPERTY_COMMENT_CLOSE	comment closing symbol	any

ENVIRONMENT VARIABLES

The following environment variables are used:

$CADP Needed. This variable contains the path of directory where CADP is installed.

$CADP/com
This directory should be put in the **$PATH** variable.

$SVL Optional. The first action of the generated script is to include the file **$CADP/src/svl/standard**, containing a list of predefined shell functions and variables. However, if the environment variable **SVL** is defined, the included file is **$SVL/src/svl/standard**. Moreover, the kernel program **`svl_kernel`** will be searched in **$SVL/bin.‘arch‘** instead of **$CADP/bin.‘arch‘**.

HOW TO READ AN SVL FILE

The tool **svl**(LOCAL) reads and processes SVL files.

BIBLIOGRAPHY

[GL01] Hubert Garavel and Frederic Lang. SVL: a Scripting Language for Compositional Verification. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee (editors), Proceedings of the 21st International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea), IFIP Conference Proceedings volume 197, pages 377-394, Kluwer, August 2001.

Available from http://cadp.inria.fr/publications/Garavel-Lang-01.html

[GLM15] Hubert Garavel, Frederic Lang, and Radu Mateescu. Compositional Verification of Asynchronous Concurrent Systems using CADP. Acta Informatica, Special Issue on Combining Compositionality and Concurrency: Part 2, 52(4-5):337-392, 2015. Available from http://cadp.inria.fr/publications/Garavel-Lang-Mateescu-15.html

[KM97] Jean-Pierre Krimm and Laurent Mounier. Compositional State Space Generation from LOTOS Programs. In Ed Brinksma (editor), Proceedings of TACAS'97 Tools and Algorithms for the Construction and Analysis of Systems (University of Twente, Enschede, The Netherlands), Lecture Notes in Computer Science volume 1217, Springer, April 1997. Available from http://cadp.inria.fr/publications/Krimm-Mounier-97.html

[Lan02] Frederic Lang. Compositional Verification using SVL Scripts. In Joost-Pieter Katoen and Perdita Stevens (editors), Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems TACAS'2002 (Grenoble, France), Lecture Notes in Computer Science volume 2280, pages 465-469, Springer, April 2002. Available from http://cadp.inria.fr/publications/Lang-02.html

SEE ALSO

aldebaran(LOCAL), **aut**(LOCAL), **bcg**(LOCAL), **bcg_cmp**(LOCAL), **bcg_graph**(LOCAL), **bcg_io**(LOCAL), **bcg_labels**(LOCAL), **bcg_min**(LOCAL), **bcg_open**(LOCAL), **caesar**(LOCAL), **caesar.adt**(LOCAL), **caesar_hide_1**(LOCAL), **caesar_rename_1**(LOCAL), **evaluator**(LOCAL), **evaluator3**(LOCAL), **evaluator4**(LOCAL), **evaluator5**(LOCAL), **exhibitor**(LOCAL), **exp**(LOCAL), **exp.open**(LOCAL), **generator**(LOCAL), **lnt.open**(LOCAL), **lotos.open**(LOCAL), **mcl**(LOCAL), **mcl3**(LOCAL), **mcl4**(LOCAL), **mcl5**(LOCAL), **projector**(LOCAL), **reductor**(LOCAL), **regexp**(LOCAL), **seq**(LOCAL), **seq.open**(LOCAL), **svl**(LOCAL), **xtl**(LOCAL), **xtl-lang**(LOCAL)

Directives for installation are given in files **$CADP/INSTALLATION_***.

Recent changes and improvements to this software are reported and commented in file **$CADP/HISTORY**.

BUGS

Please report any bug to cadp@inria.fr