User's Guide to ReaVer
version 0.9

Peter Schrammel

July 30, 2012

ReaVer[1], REActive system VERifier, is a safety verification tool for logico-numerical discrete and hybrid systems based on abstract interpretation. Its main feature is that it provides logico-numerical analysis and state space partitioning methods which enable a tradeoff between precision and efficiency. It primarily targets synchronous data flow languages like Lustre.

ReaVer is also a tool framework which makes it possible to add analysis methods and connect it to other languages.

This user guide is structured as follows:

1. Getting started
2. Framework
3. Input formats
4. Options
5. Output

For installation issues we refer to the information available on the ReaVer web site: `http://pop-art.inrialpes.fr/people/schramme/reaver/`

1 Getting started

We analyze the following small example program:

```
let node main i = (assert,ok) where
  rec assert = true
  and ok = true fby (ok && -10<=x && x<=10)
  and x = 0 fby (if i then -x else if x<=9 then x+1 else x)
```

We launch the analyzer

```
reaver example.ls
```

and we get the output (compressed):

```
[0.020] INFO [Main] ReaVer, version 0.9.0
[0.028] INFO [Main] variables(bool/num):  state=(2/1), input=(1/0)
[0.038] INFO [Verif] CFG (3 location(s), 3 arc(s)):
LOC -1:  arcs(in/out/loop)=(0,1,0), def = init
LOC -3:  arcs(in/out/loop)=(1,0,0), def = not init and not p1_
LOC -4:  arcs(in/out/loop)=(1,1,1), def = not init and p1_
[0.039] INFO [Verif] analysis 'forward analysis with abstract acceleration'
[0.070] INFO [VerifUtil] analysis result:
LOC -1:  reach =  (init) and top
LOC -3:  reach = bottom
LOC -4:  reach =  (not init and p1_) and [|-p2_+10>=0; p2_+10>=0|]
[0.074] INFO [Main] variable mapping:
"p2_" in File "example.ls", line 4, characters 17-55:
>   and x = 0 fby (if i then -x else if x<=9 then x+1 else x)
>                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
"p1_" in File "example.ls", line 3, characters 21-42:
```

---

[1] ReaVer is distributed under the Gnu GPL. For details, please refer to the LICENSE file in the distribution.

```
>   and ok = true fby (ok && -10<=x && x<=10)
>                      ^^^^^^^^^^^^^^^^^^^^^
[0.075] INFO [Main] PROPERTY TRUE (final unreachable)
```

This tells us that
- the program has two Boolean state variables and one numerical state variable and one Boolean input variable.
- After partitioning the CFG has three locations with the displayed location definitions.
- We analyzed the program using forward abstract acceleration and we obtained the displayed invariants in the locations.
- The variables occurring in the invariants correspond to the expressions in the source program listed after `variable mapping`.
- The analysis concluded with the result `PROPERTY TRUE`.

We can also display the CFG (using the Dot format, see Fig. 1).

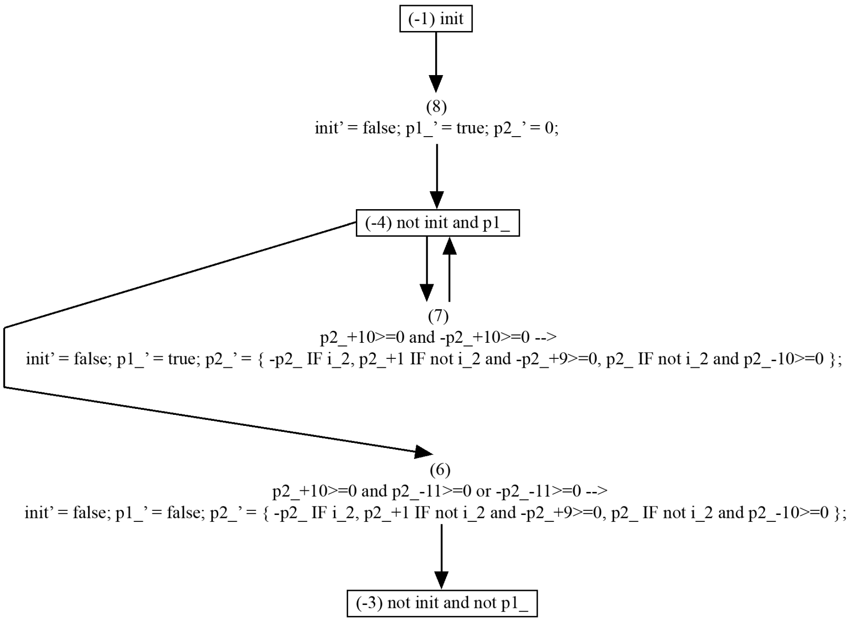

Figure 1: CFG printed to Dot: The locations are labeled with their location definitions. The arcs are labeled with "arc assertion → transition function".

2 Framework

The verification engine of ReaVer is based on a generic framework. Having a basic understanding of the structure and mode of operation is helpful for using ReaVer efficiently.

The framework provides three basic data structures:
- The *data-flow (DF) program* is the common intermediate representation of a discrete or hybrid input program.
- The *control-flow graph (CFG)* is the common representation used during analysis.

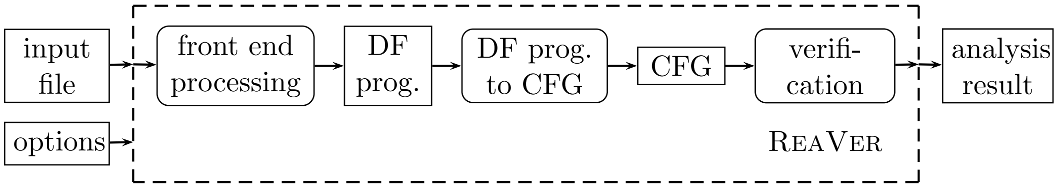

Figure 2: ReaVer: data and operation flow.

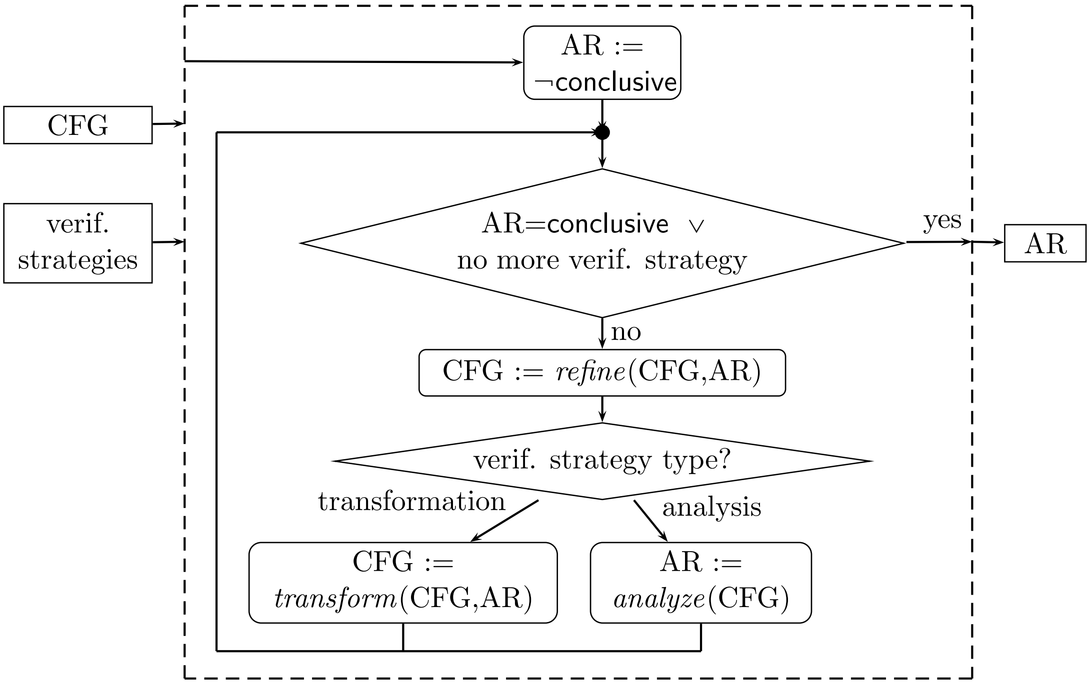

Figure 3: ReaVer: Zoom into the *verification* block of Fig. 2.

- The *analysis result (AR)*, which holds for each CFG location the corresponding (logico-numerical) abstract value.

Moreover, it defines five interfaces for the operations and modules involved in the verification process:

- *Front ends* convert an input file into a DF program.
- *DF program to CFG transformations* convert a DF program into a CFG.
- *CFG transformations* transform a CFG.
- *Analyses* analyze a CFG and produce an analysis result.
- *Abstract domains* are used by the analyses.

The tool provides implementations to these interfaces and uses them to perform the flow of operations depicted in Fig. 2: The tool takes an input file and options, the *front end* parses the input file and transforms the program do a *DF program*. The *DF program to CFG transformation* converts it into a *CFG*. Then the *verification block* (Fig. 3) provided by the framework performs the actual verification: it is given a sequence of *verification strategies* (via the tool options), *i.e. CFG transformations* and *analyses* and executes them iteratively until all verification strategies have been processed or a conclusive *analysis result* has been obtained.

3 Input formats

ReaVer currently supports the following input formats:

NBac and Hybrid NBac. The typical file extension is `.nbac`. The Hybrid NBac grammar is listed in Fig. 4; the expressions are those allowed by the BddApron library (see Table 1); besides the type definitions of enumerated types, the available types are `bool`, `real`, `int` and signed (`sint[`*n*`]`) and unsigned (`uint[`*n*`]` bounded integers represented by n bits.

A discrete (NBac format) program obeys the same syntax except that $\langle \mathit{conttrans} \rangle$ and the `up` operator are not allowed.

The property is specified by the expressions following the keywords `assertion` $\mathcal{A}$ and `invariant` $\mathcal{G}$ (or alternatively by the error states: `final` $\mathcal{E}$).

Lustre programs (`.lus`) can be converted to NBac format using the tool Lus2NBac.

Subset of Lucid Synchrone and Zelus. The typical file extension is `.ls`. The grammar can be found in Table 2.

The top-level function must have two Boolean outputs `(assert,ok)` which correspond to the two outputs $(\mathcal{A}, \mathcal{G})$ of the observer specifying the property.

The corresponding NBac/Hybrid NBac program can be printed using the option `-nbac` *filename*.

Boolean expressions:

$$\langle \mathit{Bexpr} \rangle ::= \mathsf{tt} \mid \mathsf{ff} \mid \langle \mathit{Bvar} \rangle \mid \neg\langle \mathit{Bexpr} \rangle \mid \langle \mathit{Bexpr} \rangle\ (\wedge \mid \vee \mid \ldots)\ \langle \mathit{Bexpr} \rangle$$
$$\mid\ \ \langle \mathit{expr} \rangle = \langle \mathit{expr} \rangle \mid \langle \mathit{Iexpr} \rangle\ (< \mid <=)\ \langle \mathit{Iexpr} \rangle \mid \langle \mathit{Acons} \rangle$$

Arithmetic expressions:

$$\langle \mathit{Aexpr} \rangle ::= \mathit{cst} \mid \langle \mathit{Avar} \rangle \mid (- \mid \surd)\langle \mathit{Aexpr} \rangle \mid \langle \mathit{Aexpr} \rangle\ (+ \mid - \mid * \mid / \mid \%)\ \langle \mathit{Aexpr} \rangle$$
$$\mid\ \ \text{if } \langle \mathit{Bexpr} \rangle \text{ then } \langle \mathit{Aexpr} \rangle \text{ else } \langle \mathit{Aexpr} \rangle$$

Arithmetic conditions:

$$\langle \mathit{Acons} \rangle ::= \langle \mathit{Aexpr} \rangle\ (< \mid <=)\ \langle \mathit{Aexpr} \rangle$$

Enumerated types:

$$\langle \mathit{Eexpr} \rangle ::= \mathit{label} \mid \langle \mathit{Evar} \rangle \mid \text{if } \langle \mathit{Bexpr} \rangle \text{ then } \langle \mathit{Eexpr} \rangle \text{ else } \langle \mathit{Eexpr} \rangle$$

Bounded integers:

$$\langle \mathit{Iexpr} \rangle ::= \langle \mathit{cst} \rangle \mid \langle \mathit{Ivar} \rangle \mid \langle \mathit{Iexpr} \rangle\ (+ \mid - \mid *)\ \langle \mathit{Iexpr} \rangle \mid \langle \mathit{Iexpr} \rangle\ (<< \mid >>)\ \mathrm{n}$$
$$\mid\ \ \text{if } \langle \mathit{Bexpr} \rangle \text{ then } \langle \mathit{Iexpr} \rangle \text{ else } \langle \mathit{Iexpr} \rangle$$

Expressions:

$$\langle \mathit{expr} \rangle ::= \langle \mathit{Bexpr} \rangle \mid \langle \mathit{Eexpr} \rangle \mid \langle \mathit{Iexpr} \rangle \mid \langle \mathit{Aexpr} \rangle$$

Table 1: Expressions available in BddApron (subset).

$\langle \mathit{decl} \rangle ::= \langle \mathit{typedecl} \rangle \mid \langle \mathit{fundecl} \rangle \mid \langle \mathit{decl} \rangle\ \langle \mathit{decl} \rangle$

$\langle \mathit{typedecl} \rangle ::= \texttt{type}\ t\ \texttt{=}\ L\ \texttt{|}\ \dots\ \texttt{|}\ L$

$\langle \mathit{fundecl} \rangle ::= \texttt{let}\ [\texttt{node} \mid \texttt{hybrid}]\ f\ [\langle \mathit{pat} \rangle]\ \texttt{=}\ \langle \mathit{expr} \rangle$

$\langle \mathit{pat} \rangle ::= v \mid \texttt{(}\langle \mathit{pat} \rangle\texttt{,}\dots\texttt{,}\langle \mathit{pat} \rangle\texttt{)}$

$\langle \mathit{expr} \rangle ::= v \mid \mathit{cst} \mid \mathit{op}\ \langle \mathit{expr} \rangle \mid f\ \langle \mathit{expr} \rangle \mid \texttt{(}\langle \mathit{expr} \rangle\texttt{,}\dots\texttt{,}\langle \mathit{expr} \rangle\texttt{)}$

$\quad \mid\ \langle \mathit{expr} \rangle\ \texttt{fby}\ \langle \mathit{expr} \rangle \mid \langle \mathit{expr} \rangle\ \texttt{->}\ \langle \mathit{expr} \rangle \mid \texttt{pre}\ \langle \mathit{expr} \rangle \mid \texttt{last}\ v \mid \texttt{up}\ \langle \mathit{expr} \rangle \mid \texttt{init}$

$\quad \mid\ \langle \mathit{expr} \rangle\ \texttt{on}\ \langle \mathit{expr} \rangle \mid \texttt{let}\ [\texttt{rec}]\ \langle \mathit{equ} \rangle\ \texttt{in}\ \langle \mathit{expr} \rangle$

$\langle \mathit{equ} \rangle ::= v\ \texttt{=}\ \langle \mathit{expr} \rangle \mid \langle \mathit{equ} \rangle\ \texttt{and}\ \langle \mathit{equ} \rangle$

$\quad \mid\ \texttt{der}\ v\ \texttt{=}\ \langle \mathit{expr} \rangle\ \texttt{init}\ \langle \mathit{expr} \rangle\ \texttt{reset}\ \langle \mathit{res} \rangle$

$\quad \mid\ v\ \texttt{=}\ \langle \mathit{res} \rangle\ \texttt{init}\ \langle \mathit{expr} \rangle$

$\langle \mathit{res} \rangle ::= \langle \mathit{expr} \rangle\ \texttt{every}\ \langle \mathit{expr} \rangle\ \texttt{|}\ \dots\ \texttt{|}\ \langle \mathit{expr} \rangle\ \texttt{every}\ \langle \mathit{expr} \rangle$

Table 2: Zelus syntax (subset).

$\langle \mathit{prog} \rangle ::= [\texttt{typedef}\ \langle \mathit{typedef} \rangle^+]\ \langle \mathit{vardecl} \rangle\ [\texttt{definition}\ \langle \mathit{definition} \rangle^+]$

$\quad \texttt{transition}\ \langle \mathit{transition} \rangle^+$

$\quad \langle \mathit{initial} \rangle\ [\langle \mathit{assertion} \rangle]\ \langle \mathit{invariant} \rangle$

$\langle \mathit{typedef} \rangle ::= \mathit{type}\ \texttt{=}\ \texttt{enum\{}\ \langle \mathit{labels} \rangle\ \texttt{\};}$

$\langle \mathit{labels} \rangle ::= \mathit{label} \mid \mathit{label}\ \texttt{,}\ \langle \mathit{labels} \rangle$

$\langle \mathit{vardecl} \rangle ::= \texttt{state}\ \langle \mathit{varstype} \rangle^+\ [\texttt{input}\ \langle \mathit{varstype} \rangle^+]\ [\texttt{local}\ \langle \mathit{varstype} \rangle^+]$

$\langle \mathit{varstype} \rangle ::= \langle \mathit{vars} \rangle\ \texttt{:}\ \mathit{type}\ \texttt{;}$

$\langle \mathit{vars} \rangle ::= v \mid v\ \texttt{,}\ \langle \mathit{vars} \rangle$

$\langle \mathit{definition} \rangle ::= v\ \texttt{=}\ \langle \mathit{expr} \rangle\ \texttt{;}$

$\langle \mathit{transition} \rangle ::= \langle \mathit{disctrans} \rangle \mid \langle \mathit{conttrans} \rangle$

$\langle \mathit{disctrans} \rangle ::= v\texttt{'}\ \texttt{=}\ \langle \mathit{expr} \rangle\ \texttt{;}$

$\langle \mathit{conttrans} \rangle ::= \texttt{.}v\ \texttt{=}\ \langle \mathit{expr} \rangle\ \texttt{;}$

$\langle \mathit{expr} \rangle ::= \langle \mathit{BddApronExpr} \rangle \mid \texttt{up}\ \langle \mathit{expr} \rangle$

$\langle \mathit{initial} \rangle ::= \texttt{initial}\ \langle \mathit{expr} \rangle\ \texttt{;}$

$\langle \mathit{assertion} \rangle ::= \texttt{assertion}\ \langle \mathit{expr} \rangle\ \texttt{;}$

$\langle \mathit{invariant} \rangle ::= \texttt{invariant}\ \langle \mathit{expr} \rangle\ \texttt{;} \mid \texttt{final}\ \langle \mathit{expr} \rangle\ \texttt{;}$

Figure 4: Hybrid NBac format.

3.1 Options

ReaVer is launched using: `reaver` ⟨*filename*⟩ [*options*]

The available *options* and their default values are described in this section.

Preprocessing DF program to CFG. The following options control the translation from a certain type of DF programs to CFGs:

`-p` *progtype*[`:`⟨*params*⟩]	program type of the DF program `d`... discrete program `h`... hybrid program with zero-crossings, parameters: `d=`⟨*sem*⟩ semantics of discrete zero-crossings `c=`⟨*sem*⟩ semantics of continuous zero-crossings ⟨*sem*⟩ ::= `AtZero` | `Contact` | `Crossing` default: `d=Contact,c=Contact`
`-p_help`	print the available program types

The default method is chosen based on the fact whether the set of ODEs in the DF program is empty (`d`) or not (`h`).

Verification strategies. The verification process is specified by a sequence of verification strategies, *i.e.* CFG transformations and analyses.

`-s` ⟨*strategies*⟩	use the given verification strategies
`-s_help`	print the available verification strategies

Verification strategies have the structure defined in Table 3. Table 5 and Table 6 list the available CFG transformations and analyses respectively.

Abstract domains. Analysis methods can be parametrized by an abstract domain (`d=`*dom*). The available domains are listed in Table 4. The structure of *dom* follows the ⟨*element*⟩ rule in Table 3.

`-dom_help`	print the available abstract domains

Examples for verification strategies. Table 7 lists the default verification strategies and some typical, alternative verification strategies.

Logging. The following options control the logging (debugging) output:

`-cudd_print_limit` *n*	up to which BDD size formulas are printed
`-debug` ⟨*level*⟩	debugging output verbosity ⟨*level*⟩ ::= `ERROR` | `WARN` | `INFO` | `DEBUG`[*n*] default: `INFO`
`-debug_force` ⟨*level*⟩	force debugging output verbosity (overrides maximum verbosity defined in modules)

Additional output. The following options allow to print DF programs and CFGs to certain output formats:

`-dot` *filename* `-dot_noarcs`	print CFG to Dot file option for `-dot`: do not print transition formulas
`-nbac` *filename*	print DF program in Hybrid NBac format

⟨*strategies*⟩ ::= ⟨*element*⟩ | ⟨*element*⟩ `;` ⟨*strategies*⟩
⟨*element*⟩ ::= *identifier* [`:` ⟨*params*⟩]
⟨*params*⟩ ::= ⟨*param*⟩ | ⟨*param*⟩ `,` ⟨*params*⟩
⟨*param*⟩ ::= *identifier* |*identifier* `=` *value* | *identifier* `=` '{' ⟨*element*⟩ '}'

Table 3: Structure of the option argument for specifying verification strategies

`P`	convex polyhedra	
	`l`	non-strict inequalities (default: strict inequalities)
	`p`	logico-numerical power domain (default: product domain)
`O`	octagons	
	`p`	logico-numerical power domain (default: product domain)
`I`	intervals	
	`p`	logico-numerical power domain (default: product domain)
`TE`	template emulation	
	`p`	logico-numerical power domain (default: product domain),
	`t=`{⟨*exprlist*⟩}	template given by the comma-separated list of arithmetic expressions ⟨*exprlist*⟩. Shortcuts for specifying templates: `INT`... intervals, `ZONE`... zones, `OCT`... octagons (default).
`FdpI`	finitely disjunctive partitioned interval domain	
	`p`	logico-numerical power domain (default: product domain)

Table 4: Abstract domains.

General partitioning:

`pIF`	partition by initial, final and other states
`pE`	enumerate Boolean states `v={`⟨*varlist*⟩`}` enumerate only the states of the variables given by the comma-separated list ⟨*varlist*⟩ (default: all variables)
`pM`	partition manually by the given list of splitting predicates `e={`⟨*exprlist*⟩`}` list of splitting predicates

Discrete partitioning:

`pMD`	partition by discrete numerical modes `f=`[`bi`|`bic`] forget Boolean inputs (`bi`) or Boolean inputs and numerical constraints (`bic`, default)
`pB`	refine partition by Boolean backward bisimulation

Hybrid partitioning:

`pMHB`	partition by Boolean-defined continuous modes `f=`[`bi`|`bic`] forget Boolean inputs (`bi`) or Boolean inputs and numerical constraints (`bic`, default)
`pMHN`	partition by numerically-defined continuous modes `d=`*dom* domain (default `FdpI`)
`pQ`	enumerate state variables added during the zero-crossing translation (***q***)
`pS`	split into convex staying conditions

General preprocessing:

`rT`	refine arcs by destination location
`rB`	remove Boolean inputs (splits arcs)

Preprocessing for abstract acceleration:

`rAB`	remove Boolean inputs (only in accelerable loops), parameters: `d=`[`O`|`B`|`N`] decoupling mode: no decoupling (`O`), Boolean/accelerable decoupling (`B`, default), Boolean+non-accelerable/accelerable decoupling (`N`)
`rAS`	split non-convex numerical guards in accelerable self-loops
`rAF`	flatten accelerable self-loops
`rAD`	decouple accelerable from non-accelerable or Boolean self-loops
`rAI`	inputization for decoupled self-loops

Hybrid preprocessing:

`tR`	relationalization

Table 5: Verification strategies: transformations.

Discrete analysis:

`aB`	Boolean analysis	
	`b`	backward analysis (default: forward)
`aS`	standard analysis	
	`b`	backward analysis (default: forward)
	`d=`*dom*	abstract domain (default: `P`)
	`ws=`n	use widening after n iterations (default: 2)
	`wd=`n	n descending iterations (default: 2)
`aA`	analysis with abstract acceleration	
	`b`	backward analysis (requires enumeration) (default: forward)
	`d=`*dom*	abstract domain (default: `P`)
	`ws=`n	use widening after n iterations (default: 2)
	`aws=`n	use widening after n iterations in accelerable cycles (default: 7)
	`wd=`n	n descending iterations (default: 1)
`aM`	analysis with numerical max-strategy iteration (requires enumeration)	
	`d=`*dom*	abstract domain (default: `TE`)
`aL`	analysis with logico-numerical max-strategy iteration	
	`d=`*dom*	abstract domain (default: `TE`)

Hybrid analysis:

`aH`	hybrid time-elapse	
	`d=`*dom*	abstract domain (default: `P`)
	`ws=`n	use widening after n iterations (default: 2)
	`wd=`n	n descending iterations (default: 2)
`aHM`	analysis with hybrid max-strategy iteration (requires enumeration)	
	`d=`*dom*	abstract domain (default: `TE`)
`aHL`	analysis with logico-numerical hybrid max-strategy iteration	
	`d=`*dom*	abstract domain (default: `TE`)

Table 6: Verification strategies: analyses.

Discrete programs:

Standard analysis with convex polyhedra	`aB;aB:b;pIF;pMD;rT;aS`
Standard analysis with logico-numerical octagon power domain, delayed widening by 3 and 1 descending iteration:	`aB;aB:b;pIF;pMD;rT;aS:d={O:p},ws=3,wd=1`
Abstract acceleration of enumerated CFG:	`aB;aB:b;pIF;pE;rT;rAB;rAS;rAF;rAD;aA`
Logico-numerical abstract acceleration (default):	`aB;aB:b;pIF;pMD;rT;rAB;rAS;rAF;rAD;aA`
Max-strategy iteration with octagonal templates:	`aB;aB:b;pIF;pE;rT;rB;sA;aM`
Logico-numerical max-strategy iteration with a given template:	`aB;aB:b;pIF;pMD;rT;aL:d={TE:t={-x,x+y,x-2*y}}`

Hybrid programs:

Polyhedral time-elapse:	`aB;aB:b;pIF;pE;pS;rT;rB;sA;aH`
Logico-numerical polyhedral time-elapse (default):	`aB;aB:b;pIF;pMHB;pQ;rT;aH`
Hybrid numerical max-strategy iteration with interval constraints:	`aB;aB:b;pIF;pE;pS;rT;rB;sA;aHM:d={TE:t=INT}`
Logico-numerical hybrid max-strategy iteration with zonal constraints:	`aB;aB:b;pIF;pMHB;pQ;rT;aHL:d={TE:t=ZONE}`
Logico-numerical relationalization and analysis by standard analysis with convex polyhedra and delayed widening by 5	`aB;aB:b;pIF;pMHB;pQ;tR;rT;aS:ws=5`

Table 7: Examples of typical verification strategies.

4 Output

The program output has the format of a log file: [*timestamp*] *log level* [*module*] *message*. The following information can be found in log level `INFO`:

- The number of variables: Boolean and numerical state variables, Boolean and numerical input variables:

  ```
  variables(bool/num):  state=(3/1), input=(1/0)
  ```

- The expressions for initial states, error states and the assertion:

  ```
  initial:  init
  error:  not init and not p1_
  assertion:  true
  ```

- The transformations performed and the size of CFG in number of locations and number of arcs:

  ```
  transform 'partitioning initial, final and other states'
  CFG (3 location(s), 3 arc(s)):
  LOC -1:  arcs(in/out/loop)=(0,1,0), def = init
  LOC -3:  arcs(in/out/loop)=(1,0,0), def = not init and not p1_
  LOC -4:  arcs(in/out/loop)=(1,1,1), def = not init and p1_
  ```

- The analyses performed and the computed invariants for each location:

  ```
  analysis 'forward analysis with abstract acceleration'
  analysis result:
  LOC -1:  reach =  (init) and top
  LOC -3:  reach = bottom
  LOC -4:  reach =  (not init and p1_) and [|-p2_+10>=0; p2_+10>=0|]
  analysis 'forward analysis with abstract acceleration' returned true
  ```

  In case of an inconclusive analysis (`...returned false`) the locations overlapping with the error states are marked accordingly:

  ```
  analysis 'boolean forward analysis'
  analysis result:
  LOC 0:  CONTAINS ERROR STATES, reach =  (true) and top
  analysis 'boolean forward analysis' returned false
  ```

- The final analysis result: Either the property has been verified (`PROPERTY TRUE (final unreachable)`), the property has been falsified (`PROPERTY FALSE`) or the result is inconclusive (`PROPERTY DON'T KNOW (final reachable)`).
- Variable mappings: for Zelus/Lucid Synchrone programs the correspondences between variables in the analysis result and expressions in the original program are listed:

  ```
  "p1_" in File "example.ls", line 3, characters 21-42:
  >   and ok = true fby (ok && -10<=x && x<=10)
  >                      ^^^^^^^^^^^^^^^^^^^^^
  ```

  This means that $p1\_' = ok \wedge -10 \leqslant x \wedge x \leqslant 10$.