RegSTAB's User Manual

Version 2.0.0

`http://regstab.forge.ocamlcore.org/`

Vincent Aravantinos
vincent.aravantinos@gmail.com
Google* "Vincent Aravantinos"

May 3, 2012

Contents

*The address of my website depends on the institution that employs me, since this can change a lot, I'd rather not put a concrete URL here.

1 Description

In short. RegSTAB is a SAT-solver extended to handle formula schemata i.e., constructions of the form $\bigwedge_{i=1}^{n}(\neg P_i \vee P_{i+1})$. Such schemata are considered to be unsatisfiable iff all propositional formulae of the corresponding form are unsatisfiable.

Restrictions. It is generally not possible to automatize the (un)satisfiability of such objects [Tableaux'09]. So RegSTAB is restricted to a specific form of schemata called "regular schemata", hence the part "Reg" of RegSTAB. The restrictions are detailed in Section 4.2.

How does it do it. RegSTAB is based on an extension of propositional tableaux called STAB. Hence the part "STAB" of RegSTAB (for Schematic TABleaux). Regular schemata and STAB are described in detail in [Tableaux'09]. A detailed overview of RegSTAB is provided in [IJCAR'10b].

Model and proof generation. RegSTAB is able to provide a model when the schema happens to be satisfiable, and a refutation when it is unsatisfiable. Two formats are available for proofs, both are generated in XML. The corresponding DTD files are provided with the distribution, their description is detailed in Section 5.2.

Complexity. Notice finally that the complexity of RegSTAB[1] is studied in [LATA'10]: in the (very) worst case, RegSTAB terminates in time and space $O(2^{2^n})$ where n is the size of the formula.

[1] This is not actually RegSTAB but a procedure very close, so that the results also apply to RegSTAB

Why not DPLL? This is quite unusual to use propositional tableaux for a SAT-solver but this is much more natural to use tableaux rather than DPLL to handle schemata (though this is done in [IJCAR'10a]). As a pure SAT-solver RegSTAB is all the least efficient. But one can easily think of combining RegSTAB with an efficient SAT-solver in order to benefit of both worlds.

2 Install

2.1 From sources

Steps:

1. `make all`

   Compile the byte-code version of RegSTAB. This generates the executables `regstab`, `sch2cnf`, `sch2ltl`.

2. (Optional) `make opt`

   Compile the native-code version of RegSTAB (which is much faster). This is possible only if Ocaml native-code compilation is possible on your machine, see the Ocaml manual `http://caml.inria.fr/ocaml/portability.en.html`. Most architectures are normally supported. This generates the executable `regstab.opt`, `sch2cnf.opt`, `sch2ltl.opt`.

3. `make install` (as root)

   - Copy the files `regstab`, `regstab.opt` (if any), `sch2cnf`, `sch2cnf.opt` (if any), `sch2ltl`, and `sch2ltl.opt` (if any) into the directory `$PREFIX/bin/`.
   - Copy the manual (this file) into the directory `$PREFIX/doc/regstab/`.
   - Copy the man pages into the directory `$PREFIX/man/man1/`.
   - Copy the developer doc into the directory `$PREFIX/share/regstab/developper_doc/`.
   - Copy the proof DTD files into the directory `$PREFIX/share/regstab/DTD/`.
   - Copy the vim syntax file into the directories `$HOME/.vim/syntax` and `$PREFIX/share/regstab/vim/`.

   The environment variable `PREFIX` defaults to `/usr/local`.

Dependencies.

- Ocaml 3.12 or above[2] (yes, 3.12 is really needed due in particular to the use of first-class modules).
- For Windows: MinGW[3].

[2] `http://caml.inria.fr/index.en.html`
[3] `http://www.mingw.org/`

2.2 Win32

The Win32 archive (not always available, I do my best as I do not have a Win32 machine) contains the following:

- `QUICKSTART`: "Short manual".
- `bin/`: Contains `regstab.exe`, `regstab.opt.exe`, `sch2cnf.exe`, `sch2cnf.opt.exe`. Files with `.opt` are Win32 native executables, other files are machine-independent bytecode executables. *Note that all these executables shall be run in the Windows (DOS-like) command-line.* Take care: ending the standard input is done by CTRL+Z (and not CTRL+D as on Unix).
- `doc/`: Contains the manual `manual.pdf` (this file).
- `examples/`: Contains examples.
- `man/man1/`: Contains the man pages for `regstab`, `regstab.opt`, `sch2cnf`, `sch2cnf.opt`
- `tools/`: Contains `regstab.vim` the vim syntax file for RegSTAB files, and the DTD files for proofs and schemata.

2.3 Intel Mac OSX Binaries

The Intel Mac OSX archive contains the following:

- `QUICKSTART`: "Short manual".
- `bin/`: Contains `regstab`, `regstab.opt`, `sch2cnf`, `sch2cnf.opt`. Files with suffix `.opt` are Intel Mac OSX native executables, files without suffix are machine-independent bytecode executables.
- `doc/`: Contains the manual `manual.pdf` (this file).
- `examples/`: Contains examples.
- `man/man1/`: Contains the man pages for `regstab`, `regstab.opt`, `sch2cnf`, `sch2cnf.opt`.
- `tools/`: Contains `regstab.vim` the vim syntax file for RegSTAB files, and the DTD files for proofs and schemata.

2.4 Machine-Independent Bytecode

*Warning: the bytecode version of Reg*STAB *is much slower than the native one. Though you may have no other choice if your architecture is not supported by the OCaml native compiler (very rare).*

The Bytecode archive contains the following:

- `QUICKSTART`: "Short manual".

- `bin/`: Contains `regstab` and `sch2cnf`.
- `doc/`: Contains the manual `manual.pdf` (this file).
- `examples/`: Contains examples.
- `man/man1/`: Contains the man pages for `regstab` and `sch2cnf`.
- `tools/`: Contains `regstab.vim` the vim syntax file for RegSTAB files, and the DTD files for proofs and schemata.

2.5 GODI

Note: there are merely no dependencies so it is very easy to install RegSTAB without GODI.

GODI[4] is a package manager for Ocaml libraries and software. It has many many advantages for Ocaml apps developpers.

See GODI documentation and install the package "apps-regstab". The following will be installed (`<PREFIX>` is GODI base directory):

- `regstab`, `regstab.opt` (if any), `sch2cnf`, `sch2cnf.opt` in `<PREFIX>/bin`.
- The manual (this file) into `<PREFIX>/doc/apps-regstab/`.
- The man pages into the directory `<PREFIX>/man/man1/`.
- The developer doc into the directory `<PREFIX>/share/apps-regstab/developper_doc`.
- The proof DTD files into the directory `<PREFIX>/share/apps-regstab/DTD`.
- The vim syntax file into the directories `$HOME/.vim/syntax` and `<PREFIX>/share/apps-regstab/vim`.

3 Using the executables

RegSTAB *is always used* via *the command-line.*

`regstab.opt` [OPTIONS] [$file$]
`regstab` [OPTIONS] [$file$]

Prints `UNSATISFIABLE` (resp. `SATISFIABLE`) if the input formula is unsatisfiable (resp. satisfiable). If no file is provided, the input formula is taken on `stdin` (to send your formula type in `CTRL+D` on Unix/Linux/MacOS X, `CTRL+Z` on Windows).

[4] `http://godi.camlcity.org/godi/index.html`

Options:

--exclude-vars v1,v2,...
: If the schema is satisfiable, exclude variables **v1,v2,...** of the printed model (can improve readability of the model if you know that the values of some variables are not significant). See Section 5.1 for an example.

-l
: Print the lemmas used in the deduction (in the end only, not during execution).

--lemmas
: Print the lemmas database (all lemmas, not just those used in the deduction).

-m
: Print a model when the schema is satisfiable. See Section 5.1 for details.

--model
: Same as -m.

--not-tune-gc
: Turn off garbage collector tuning (which just amounts to increase the size of the minor heap).

-p file
: Output the proof in the given file when the schema is unsatisfiable.

--machine-proof
: When the proof is generated, generate it in a "machine"-oriented way, see Section 5.2 for details.

--no-utf8
: Display text in ASCII rather than UTF8.

--help
: Prints the list of options.

-help
: Same as --help.

--human-xml
: Output XML in such a way that it is (a bit) more readable by humans (right now it just indents the file). Sometimes more time-consuming.

--proof file
: Same as -p.

--used-lemmas
: Same as -l.

--stats
Provide statistics once the execution is over (number of rules applied, number of lemmas, ...).

-s
Same as --stats.

--use-inclusion Use inclusion instead of equality to detect cycles (generally slower but generates more concise proofs).

--verbose n
Be verbose according to the level (1-3, the higher the more verbose). Displays among other things:

- the input formula as it is parsed by RegSTAB
- rule applications
- cycle detection
- lemma additions to the database

-v n
Same as --verbose.

-x v1,v2,...
Same as --exclude-vars.

4 Input language

We start with an informal description of the language, pointing out worth noticing points. The formal grammar for schemata is given in Figure 1.

4.1 Propositional Formulae

- Usual logical notations are translated into ASCII: the conjunction ($\wedge$) is written `/\`, the disjunction ($\vee$) is written `\/`, and the negation ($\neg$) is written `~`.

  As a convenience some other usual connectives are pre-defined: the implication ($P_1 \Rightarrow P_2 := \neg P_1 \vee P_2$) is written `P_1->P_2`, the equivalence ($P_1 \Leftrightarrow P_2 := (P_1 \Rightarrow P_2) \wedge (P_2 \Rightarrow P_1)$) is written `P_1<->P_2`, and the exclusive or ($P_1 \oplus P_2 := \neg(P_1 \Leftrightarrow P_2)$) is written `P_1(+)P_2`.

- Propositional variables must be indexed: you cannot write `A/\(B\/C)` but `P_1/\(P_2\/Q_1)` is ok. They can be any alphanumerical sequence starting with an uppercase letter. Prime (`'`) can be appended to the sequence. The index may be any number.

- The precedence of connectives is as follows: `/\` > `\/` > `(+)` > `<->`, `->`.

Notice that formulae are internally translated into negation normal form, i.e., negation only occurs in front of propositional variables (after equivalence, implication and exclusive or have been turned into conjunctions and disjunctions).

4.2 Schemata

Syntax:

- Iterated conjunctions are written "`/\`i`=`k`..`e" where i is an integer variable, k is an integer, and e is an arithmetic expression. We call k the *lower bound* of the iterated conjunction, and e is its *upper bound*. Iterated disjunctions are written similarly with `\/` instead of `/\`.
- Arithmetic expressions are written "n`+`k" or "n`-`k" where n is an integer variable and k is a natural number.
- Inside iterations, indexed propositional variables are written "`P_`e" where P is a propositional variable (see 4.1) and e is an arithmetic expression. *Do not put parentheses around e.*
- Integer variables can be any alphanumerical sequence starting with a lower case letter. Prime (`'`) can be appended to the sequence.
- Iteration operators have the highest precedence: `/\i=0..n P_i/\P_i+1` is interpreted as `(/\i=0..n P_i)/\P_i+1`, and not `/\i=0..n (P_i/\P_i+1)` (think of the body of the iteration as being an argument given to the operator `/\i=0..n`).

Example: `P_1 /\ /\i=1..n-1 (P_i->P_i+1) /\~P_n`

Restrictions:

- *Iterations cannot be nested*: you cannot write `/\i=1..n (\/j=1..n ...)`
- *There may be only one free variable* (called the *parameter* of the schema): you cannot write `/\i=1..n P_i /\ /\i=2..p Q_i`.
- *All iterations must have the same bounds*[5]: you cannot write `/\i=1..n .../\ /\i=2..n ...`
- *For `P_`e occurring in some iteration, the only variable that can occur in e is the variable which is iterated*[6] you cannot write `/\i=1..n P_n+1` but `/\i=1..n P_i+1` is ok.

[5] In most cases, this can be easily circumvented, e.g., if $n > 1$ we can manually unfold the first ranks: $\bigwedge_{i=1}^{n} S_i \wedge \bigwedge_{i=2}^{n} T_i$ is equivalent, *if* $n > 1$, to $S_1 \wedge \bigwedge_{i=2}^{n} S_i \wedge \bigwedge_{i=2}^{n} T_i$

[6] This can be easily circumvented by factorizing the constant indexed proposition: $\bigwedge_{i=1}^{n}(P_n \vee P_i)$ is equivalent to $P_n \vee \bigwedge_{i=1}^{n} P_i$.

4.3 Constraints

Basic constraints can be given on the parameter of a schema. They must be inserted after the schema and are written "`|` n op k" where n is the parameter of the schema, k is an integer, and $op \in \{$`=,>=,>`$\}$.

Example:

```
P_1 /\ /\i=1..n-1 (P_i->P_i+1) /\~P_n | n > 0
```

Notice that this example is unsatisfiable with the constraint but is satisfiable without it: if we take $n = 0$ we get the formula `P_1/\~P_0` which is satisfiable. As schemata are considered to be unsatisfiable iff all propositional formulae obtained by giving a value to `n` are unsatisfiable, this schema is not satisfiable.

Restriction: positive length. Let k_1 and $n + k_2$ be the lower and upper bounds, respectively, of the iterations occurring in the schema. Then the constraint should entail $n \geq k_1 - k_2 - 1$, i.e., it should ensure that the length of iterations is positive. Concretely if we have a constraint of the form $n \geq k_3$ then we must have $k_3 \geq k_1 - k_2 - 1$.

Example:

```
P_1 /\ /\i=1..n-1 (P_i->P_i+1) /\~P_n | n > 0
```

We have here $k_1 = 1$ and $k_2 = -1$. So $n \geq k_1 - k_2 - 1$ amounts to $n \geq 1$ which is indeed entailed by $n > 0$.

The same holds for:

```
P_1 /\ /\i=1..n-1 (P_i->P_i+1) /\~P_n | n > 1
P_1 /\ /\i=1..n-1 (P_i->P_i+1) /\~P_n | n > 2
P_1 /\ /\i=1..n-1 (P_i->P_i+1) /\~P_n | n > 3
...
```

But not for:

```
P_1 /\ /\i=1..n-1 (P_i->P_i+1) /\~P_n | n > -1
P_1 /\ /\i=1..n-1 (P_i->P_i+1) /\~P_n | n > -2
P_1 /\ /\i=1..n-1 (P_i->P_i+1) /\~P_n | n > -3
...
```

Notice that this restriction could be removed but at the expense of performance.

4.4 Functions

To ease the input you can define simple functions, e.g., if you use often `A_i->A_i+1` with a different `A` (say `B_i->B_i+1`, `C_i->C_i+1`, ...), then you can factorize this by defining a function $\lambda X.X_i \Rightarrow X_{i+1}$. The syntax is "`let F(X) := X_i->X_i+1 in ...`" with the following constraints:

- The name of a function follows the same conventions as propositional variable names.

- The parameters of the function is a comma separated list comprised between parentheses if the list is non-empty. *The parameters may be either propositional variable names or simple variable names.* E.g. you can write `let F(X,n) := X_n ->X_n+1 in ...`
- The right member of the affectation is any formula as defined previously. It cannot contain a constraint.

Calling the function is done as follows: `F(P,n+1)`, i.e., the name of the function followed by the list of parameters enclosed between parentheses. *When there is no parameter, you should still put parentheses, i.e., `F()`.*

Full Example:

```
let F(S,A,B,C,i) := S_i <-> (A_i (+) B_i (+) C_i-1) in
/\i=1..n (F(S,A,B,C,i) \/ F(S',A',B',C,i+1))
```

The corresponding formal grammar is given separately in Figure 2.

4.5 Comments

Comments start by `//` and end at the end of the line.

The special comment `//expected: unsat` or `//expected: sat` can be used before the schema in order to indicate to RegSTAB whether the schema is expected to be satisfiable or not. If the result is not the expected one then RegSTAB will issue an error message of the following form: `ERROR: EXPECTED TO BE SAT` or `UNSAT`, respectively. All the files in the distribution directory `examples` make use of this feature.

5 Output

The standard output of RegSTAB is just "`SATISFIABLE`" or "`UNSATISFIABLE`", but one can use several options to generate witnesses: a model (or set of models) when the input schema is satisfiable, and a refutation when it is unsatisfiable. We now give more details on these options.

5.1 Model generation

Model generation is triggered by the command-line option **-m**: RegSTAB will then print a model when the input schema is found to be satisfiable. For instance:

```
> regstab.opt -m
P_1 /\ Q_2
SATISFIABLE
Found model:
P_1, Q_2
```

You can use the option **--exclude-vars** to exclude some variables from the output:

```
> regstab.opt -m --exclude-vars P
P_1 /\ Q_2
SATISFIABLE
Found model (variable P is excluded):
Q_2
```

When the input schema contains a variable, RegSTAB tries to provide more information:

```
> regstab.opt -m
P_1 /\ /\i=1..n (P_i->P_i+1)
SATISFIABLE
for any n≤0, the corresponding instance is:
P_1
and the following is a model:
P_1
```

Or:

```
> regstab.opt -m
Taking input from stdin.
P_1 /\ /\i=1..n (P_i->P_i+1) | n>=4
SATISFIABLE
for n=4, the corresponding instance is:
P_1 /\ (P_1->P_2) /\ (P_2->P_3) /\ (P_3->P_4) /\ (P_4->P_5)
and the following is a model:
P_5, P_4, P_3, P_2, P_1
```

5.2 Refutation generation

Refutation generation is triggered by the command-line option **-p** (or equivalently **--proof**). The name of a file in which to store the refutation should be provided right after the option as follows:

```
> regstab.opt -p refutation.xml
```

RegSTAB will then store the refutation in the file `refutation.xml` if the schema happens to be unsatisfiable. There exist two output proof formats: a "machine-oriented" one, and a "human-oriented" one (the latter being the default). Both of them are explained informally in the following sections. The formal presentation of the corresponding proof systems is not presented but is very close to, e.g., [Tableaux'09] or [LATA'10].[7]

5.2.1 Machine-oriented format

This format is called machine-oriented because it is the closest to the internal representation of RegSTAB and because it prevents, by construction, many ill-formed proofs and schemata. Thus it removes the burden of checking criteria

[7] Contact me directly in case you want more precise information.

that have already been checked by RegSTAB. This is useful if you plan to input the proof to a proof-transformation algorithm (like the tableau to resolution transformations presented in [FTP'11], or if you want to interface RegSTAB with a proof assistant). In particular, it has the following features:

- the parameter name is stored only once (since it is not supposed to change along the procedure);
- the iteration variable is not mentioned (since it is the only variable that can appear in iterations, we just need to store the *position* of the variable, not its name);
- the bounds of all iterations are the same and are preserved all along the procedure (except when unfolding an iteration but the shift can be computed if needed), so they are stored only once;
- the grammar used for schemata only accepts regular schemata (in particular, the grammar prevents the nesting of iterations);
- the proof consists of one single tree (more precisely a tableau [Tableaux'09]) with some leaves closed by "looping" to a previously seen node (this is in contrast with the human-oriented format presented in the next section, which contains independent proofs representing the proofs of the lemmas used in the recursion to prove the main result);
- the rules that are used are clear from their XML tag thus making them easier to process;
- each rule application provides the involved active formulae (or the set of removed literals in the case of the pure literal rule).

The file format is fully documented in the DTD file `tools/regularproof.dtd`.

5.2.2 Human-oriented format

This is the default output format. Contrarily to the machine-oriented format of the previous section, every element contains all the required information explicitly. It thus makes it more readable by a human, hence its name. In particular, as opposite to the machine-oriented format, it has the following features:

- the parameter name, iteration variable name, and bounds of iterations are stored in every single schema;
- the proof consists of small sequent-style proofs; when a sequent is encountered which requires itself a separate inductive proof, then this proof is done independently and a reference to this proof is stored in the original proof (this reference is called a "link").

It also has the "feature" that non-regular schemata can be expressed, so if you want to get rid of useless checks you'd better use the machine-oriented format. On the other hand it makes the file format more natural and easier to read. The file format is fully documented in the DTD file `tools/prooftrees.dtd`.

This output can be graphically visualized using Proof Tool (`http://www.logic.at/prooftool/`)[8].

6 Examples

Figure 3 presents a list of the provided examples along with an indicative time that it takes on my machine. Only unsatisfiable examples are presented (satisfiable ones are almost immediately found).

The script `examples/run` runs RegSTAB on the examples. Note that this script was written for development purposes only so you can try to use it but it depends on various libraries and tools – Findlib (`http://www.camlcity.org/archive/programming/findlib.html`), Fileutils (`https://forge.ocamlcore.org/projects/ocaml-fileutils/`), and maybe other-things-that-are-installed-on-my-computer-but-I-am-not-aware-that-they-are-needed... – and some paths used inside the script are most probably different between my machine and yours. The script automatically runs all the files in the `examples` directory and stores the resulting execution times in some file in the directory `examples/benchs`.

Most examples are schemata formalizations of some basic circuits. In addition, some examples come from the cut-elimination system CERES (`http://www.logic.at/ceres`). Finally, even more examples can be found by transforming LTL formulae into schemata using `ltl2sch` [TIME'11] (Google "Vincent Aravantinos", find the Software section).

7 Various

7.1 sch2cnf

`sch2cnf.opt -param` n [$file$]
`sch2cnf -param` n [$file$]

Computes the propositional formula obtained by giving the value n to the parameter of the input schema. Outputs the formula in DIMACS cnf format. Thus `sch2cnf` can be used as a generator of problems for SAT-solvers. If no file is provided the input formula is taken on `stdin`.

[8] Proof Tool evolves quickly so if there is a problem viewing the proofs, please contact directly the authors of Proof Tool. Also, in case of problems visualizing the proofs, try triggering on and off the UTF8 output of RegSTAB.

Options:

-cnf Forces the displayed formula to be in conjunctive normal form, only useful when `-H` is set.

-D Displays the formula in DIMACS cnf format (default).

-H Displays the formula in a human readable format.

7.2 sch2ltl

`sch2ltl.opt` [*file*]
`sch2ltl` [*file*]

If no file is provided the input formula is taken on `stdin`.

Computes an LTL formula out of a *positive* schema. Positive schemata are a strict subset of regular schemata that are equivalent to LTL formulae [TIME'11]. A detailed description of their restrictions are given in [TIME'11]. The output format is the one of `pltl` (`http://users.cecs.anu.edu.au/~rpg/PLTLProvers`). Note that there is also a converse translation, available independently as "`ltl2sch`" (Google "Vincent Aravantinos", find the Software section).

7.3 Vim syntax file

`regstab.vim`

Copy the file into `~/.vim/syntax/`. You can use modelines to force the syntax (see the provided examples), you just have to add as the last line of your file:

```
// vim:ft=regstab
```

You can also create a file `~/.vim/ftdetect/regstab.vim` just containing the following line:

```
au BufRead,BufNewFile *.stab set filetype=regstab
```

7.4 Man pages

Short man pages for quick recall are available in the directory `man`. If you do not wish to install RegSTAB you can access them with `man -M man/ regstab` or `man -M man/ sch2cnf` when in the top directory. However only the present file should be considered as an exhaustive documentation.

7.5 Developer documentation

If you are interested in the internals of RegSTAB, a set of HTML pages is available as a developer documentation. These are available on RegSTAB web page independently of RegSTAB's distribution.

7.6 Licence

This software is published under the terms of the CeCILL-B licence, found in the distribution. This licence is compatible with the BSD licence and is adapted to French legal matters. More information on the CeCILL-B licence can be found on Wikipedia `http://en.wikipedia.org/wiki/CeCILL`.

References

[Tableaux'09] Vincent Aravantinos, Ricardo Caferra, and Nicolas Peltier. A Schemata Calculus for Propositional Logic. In Martin Giese and Arild Waaler, editors, *TABLEAUX*, volume 5607 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2009.

[IJCAR'10a] Vincent Aravantinos, Ricardo Caferra, and Nicolas Peltier. A Decidable Class of Nested Iterated Schemata. In Giesl and Hähnle [IJCAR'10].

[LATA'10] Vincent Aravantinos, Ricardo Caferra, and Nicolas Peltier. Complexity of the Satisfiability Problem for a Class of Propositional Schemata. In Adrian-Horia Dediu, Henning Fernau, and Carlos Martn-Vide, editors, *Language and Automata Theory and Applications*, volume 6031 of *Lecture Notes in Computer Science*, pages 58–69. Springer, Heidelberg, 2010.

[IJCAR'10b] Vincent Aravantinos, Ricardo Caferra, and Nicolas Peltier. RegSTAB: A SAT-Solver for Propositional Iterated Schemata. In Giesl and Hähnle [IJCAR'10].

[TIME'11] Vincent Aravantinos, Ricardo Caferra, and Nicolas Peltier. Linear temporal logic and propositional schemata, back and forth. In *18th International Symposium on Temporal Representation and Reasoning, TIME 2011, Lbeck, Germany, September 12-14, 2011, Proceedings*, 2011.

[FTP'11] Vincent Aravantinos and Nicolas Peltier. Generating schemata of resolution proofs. In *First-Order Theorem Proving, 8th International Workshop, FTP 2011, Bern, Switzerland*, 2011.

[IJCAR'10] Jürgen Giesl and Reiner Hähnle, editors. *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, Scotland, July 16-19, 2010, Proceedings*, Lecture Notes in Computer Science. Springer, 2010.

sentence	::=	*schema*
	|	*schema* `|` *constraint*
schema	::=	*indexed-prop* `_` *linear-expression*
	|	*schema* `/\` *schema*
	|	*schema* `\/` *schema*
	|	*schema* `->` *schema*
	|	*schema* `(+)` *schema*
	|	*schema* `<->` *schema*
	|	`~` *schema*
	|	`(` *schema* `)`
	|	`/\` *var* `=` *integer* `..` *linear-expression* *no-iteration*
	|	`\/` *var* `=` *integer* `..` *linear-expression* *no-iteration*
no-iteration	::=	*indexed-prop* `_` *linear-expression*
	|	*no-iteration* `/\` *no-iteration*
	|	*no-iteration* `\/` *no-iteration*
	|	*no-iteration* `->` *no-iteration*
	|	*no-iteration* `(+)` *no-iteration*
	|	*no-iteration* `<->` *no-iteration*
	|	`~` *no-iteration*
	|	`(` *no-iteration* `)`
constraint	::=	*var* `<=` *integer*
	|	*var* `>=` *integer*
	|	*var* `<` *integer*
	|	*var* `>` *integer*
	|	*var* `=` *integer*
linear-expression	::=	*var*
	|	*integer*
	|	*var* `+` *integer*
	|	*var* `-` *integer*
var	::=	`a...z` $\{$`a...z|0...9|'`$\}^*$
indexed-prop	::=	`A...Z` $\{$`A...Z|a...z|0...9|'`$\}^*$
integer	::=	$\{$`0...9`$\}^+$

Figure 1: Main Grammar.

sentence	::=	`...`
	|	`let` *definition* `:=` *schema* `in` *sentence*
schema	::=	`...`
	|	*function-call*
definition	::=	*indexed-prop* `(` *parameters* `)`
	|	*indexed-prop*
parameters	::=	*indexed-prop*
	|	*var*
	|	*indexed-prop*`,` *parameters*
	|	*var*`,` *parameters*
function-call	::=	*indexed-prop* `(` *arguments* `)`
	|	*indexed-prop* `()`
arguments	::=	*linear-expression*
	|	*indexed-prop*
	|	*indexed-prop* `,` *arguments*
	|	*linear-expression* `,` *arguments*

Figure 2: Grammar extension for definitions.

Ripple-carry adder	
$x + 0 = x$	0.014s
commutativity	0.015s
associativity	0.054s
$3 + 4 = 7$	2.753s
$x + y = z_1 \wedge x + y = z_2 \Rightarrow z_1 = z_2$	0.374s
unicity of the result	0.079s
Carry-propagate adder	
$x + 0 = x$	0.013s
commutativity	0.035s
associativity	0.497s
equivalence between two different definitions of the same adder	0.029s
equivalence with the ripple-carry adder	0.040s
Comparisons between bit-vectors	
$x \geq 0$	0.012s
Symmetry of $\leq$ (i.e., $x \leq y \wedge x \geq y \Rightarrow x = y$)	0.012s
Totality of $\leq$ (i.e., $x > y \vee x \leq y$)	0.013s
Transitivity of $\leq$	0.014s
$1 \leq 2$	0.018s
Presburger arithmetic with bit vectors	
$x + y \geq x$	0.015s
$x_1 \leq x_2 \leq x_3 \Rightarrow x_1 + y \leq x_2 + y \leq x_3 + y$	4.623s
$x_1 \leq x_2 \wedge y_1 \leq y_2 \Rightarrow x_1 + y_1 \leq x_2 + y_2$	0.105s
$x_1 \leq x_2 \leq x_3 \wedge y_1 \leq y_2 \leq y_3 \Rightarrow x_1 + y_1 \leq x_2 + y_2 \leq x_3 + y_3$	1m24s
$1 \leq x + y \leq 5 \wedge x \geq 3 \wedge y \geq 4$	1m15s
Examples coming from CERES	
ex1original	0.034s
ex1 (same as ex1original, with trivial redundancies removed)	0.013s
ex2original	0.140s
ex2 (same as ex2original, with trivial redundancies removed)	0.105s
Other	
automata inclusion	0.069s
$\bigvee_{i=1}^{n} P_i \wedge \bigwedge_{i=1}^{n} \neg P_i$	0.036s
$P_1 \wedge \bigwedge_{i=1}^{n} (P_i \Rightarrow P_{i+1}) \wedge \neg P_{n+1} \mid n \geq 0$	0.032s
$P_1 \wedge \bigwedge_{i=1}^{n} (\neg P_i \vee P_{i+1}) \wedge \neg P_{n+1} \mid n \geq 0$	0.012s
model checking of some safety property	0.081s

Figure 3: Provided examples and indicative execution time.