RegSTAB's User Manual

Version 1.4.5

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

Vincent Aravantinos
vincent.aravantinos@gmail.com
http://membres-liglab.imag.fr/aravantinos

July 6, 2010

Contents

1 Description

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.

It is generally not possible to automatize the (un)satisfiability of such objects [ACP09]. So RegSTAB is restricted to a specific form of schemata called "regular schemata". Hence the part "Reg" of RegSTAB. Furthermore RegSTAB is based on an extension of propositional tableaux called STAB. Hence the part "STAB" of RegSTAB. Regular schemata and STAB are described in detail in [ACP09]. A detailed overview of RegSTAB is provided in [ACP10c].

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 [ACP10a]). 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.

Notice finally that the complexity of RegSTAB[1] is studied in [ACP10b]: in the (very) worst case, RegSTAB terminates in time and space $O(2^{2^n})$ where n is the size of the formula.

2 Install

2.1 From sources

Steps:

1. `make all`

   Compile the byte-code version of RegSTAB.

2. (Optional) `make opt`

   Compile the native-code version of RegSTAB if possible on your machine (→ executable `regstab.opt`).

3. (Optional) `make test`

   Run tests.

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

4. `make install` (as root)

   - Copy the files `regstab`, `regstab.opt` (if any), `sch2cnf`, `sch2cnf.opt` 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 vim syntax file into the directories `$HOME/.vim/syntax` and `$PREFIX/share/regstab/vim`.

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

Dependencies.

- The Ocaml compiler, tested with 3.10.2 and 3.11.1[2].
- If you're under Windows: MinGW[3].
- If you want to run tests: OUnit[4] (tested with 1.0.3) and the Findlib[5] library manager (tested with 1.2.4).

2.2 Win32

The Win32 archive (not always available, I do my best as I don't 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. *Notice that all those 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`
- `vim/`: Contains `regstab.vim` the vim syntax file for RegSTAB files

[2] `http://caml.inria.fr/index.en.html`
[3] `http://www.mingw.org/`
[4] `http://www.xs4all.nl/~mmzeeman/ocaml/`
[5] `http://projects.camlcity.org/projects/findlib.html/`

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`
- `vim/`: Contains `regstab.vim` the vim syntax file for RegSTAB files

2.4 Machine-Independant Bytecode

*Warning: the bytecode version of Reg*STAB *is much slower than the native one (*2*s vs.* 30*s for* `examples/adder4.stab` *on my machine). Though you may have no other choice: e.g. if your architecture is not supported by the OCaml native compiler (very rare) or by one of the dependencies.*

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

2.5 GODI

Note w.r.t. older versions: dependencies are now drastically reduced so it is very easy to install RegSTAB without GODI.

GODI[6] is a package manager for Ocaml libraries and software. It has many many advantages for Ocaml apps developpers.

Currently, the official version of GODI relies on ocaml 3.10, there is a beta version of GODI for ocaml 3.11.1[7]. See GODI documentation and install the package "apps-regstab". The following will be installed (`<PREFIX>` is GODI base directory):

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

[7] `http://download.camlcity.org/download/godi-rocketboost-20090421.tar.gz`

- `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 vim syntax file into the directories `$HOME/.vim/syntax` and `<PREFIX>/share/apps-regstab/vim`.

3 Usage

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/macosx, `CTRL+Z` on Windows).

Options:

--help
Prints the list of options.

-help Same as --help.

--print-lemmas
Prints the list of lemmas (in the end only, not during execution).

-l Same as --print-lemmas.

--verbose
Be verbose, currently displays:

- the input formula as it is parsed by RegSTAB
- the number of rules applications
- the number of lemmas
- the maximal number of unfoldings
- the number of closed and looping leaves

-v Same as --verbose

4 Language Definition

We start with an informal description of the language, pointing out worth noticing points. The formal grammar is given at the end of the section.

4.1 Propositional Formulae

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

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

- Propositional variables must be indexed: you can't 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 integer.

- Formulae must be in negative normal form i.e. negation can only occur just in front of a propositional variable: you can't write `~(P_1/\P_2)` but `~P_1\/~P_2` is ok.

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

4.2 Schemata

Syntax:

- Iterated conjunctions are written "`/\`i`=`k`..`e" where i is a variable, k is an integer, and e is an arithmetic expression. k is called the *lower bound* of the iterated conjunction, 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 a variable and k is a natural number.

- Inside iterations indexed propositional variables are written "P`_`e" where P is a propositional variable (defined in 4.1) and e is an arithmetic expression. *Do not put parentheses around* e.

- 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*[8]: 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*[9] you cannot write `/\i=1..n P_n+1` but `/\i=1..n P_i+1` is ok.

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 $op \in \{$`=`,`>=`,`>`$\}$. *Warning: since version 1.4.5 we do not allow constraints of the form* $n < k$ *or* $n <= k$ *anymore.*

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
```

[8] 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 $S_1 \wedge \bigwedge_{i=2}^{n} S_i \wedge \bigwedge_{i=2}^{n} T_i$

[9] This can be easily circumvented by factorising the constant indexed proposition: $\bigwedge_{i=1}^{n}(P_n \vee P_i)$ is equivalent to $P_n \vee \bigwedge_{i=1}^{n} P_i$. Maybe we should automatize this.

```
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 bad performance. Inform me if you feel like it is an important lacking feature.

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 $\lambda X \cdot X_i \Rightarrow X_{i+1}$. The syntax is as follows:
`let F(X) := X_i->X_i+1 in ...`

- 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, e.g., 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))
```

4.5 Comments

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

4.6 Formal Grammar

The main formal grammar is given in figure 1. The grammar for the definition of functions as described in Section 4.4 is given separately in figure 2.

5 Examples

Figure 3 presents a list of the provided examples (not necessarily up to date) along with an indicative time that it takes on my machine. All of those can be found in the directory `examples`.

6 Tools

6.1 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`.

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

6.2 Vim syntax file

`regstab.vim`

Copy the file into `~/.vim/syntax/`. You can use modelines to force the syntax (see 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
```

6.3 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 the full documentation is the present file.

7 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

[ACP09] 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.

[ACP10a] Vincent Aravantinos, Ricardo Caferra, and Nicolas Peltier. A Decidable Class of Nested Iterated Schemata. In Giesl and Hähnle [GH10].

[ACP10b] 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 - To appear, 2010.

[ACP10c] Vincent Aravantinos, Ricardo Caferra, and Nicolas Peltier. RegSTAB: A SAT-Solver for Propositional Iterated Schemata. In Giesl and Hähnle [GH10].

[GH10] Jrgen 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 - To appear, 2010.

```
          sentence  ::=  schema
                     —   schema | constraint
            schema  ::=  indexed-prop _ linear-expression
                     —   ~ indexed-prop _ linear-expression
                     —   schema /\ schema
                     —   schema \/ schema
                     —   schema -> schema
                     —   schema (+) schema
                     —   schema <-> schema
                     —   ( schema )
                     —   /\ var = integer .. linear-expression no-iteration
                     —   \/ var = integer .. linear-expression 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	
$x + 0 = x$	0.002s
commutativity	0.016s
associativity	2.219s
$3 + 4 = 7$	0.197s
Carry-propagate adder	
$x + 0 = x$	0.002s
commutativity	0.008s
associativity	0.375s
equivalence between two different definitions of the same adder	0.008s
equivalence with the ripple-carry adder	0.011s
Comparisons between bit-vectors	
$x \geq 0$	0.001s
Symmetry of $\leq$ (i.e. $x \leq y \wedge x \geq y \Rightarrow x = y$)	0.001s
Totality of $\leq$ (i.e. $x > y \vee x \leq y$)	0.001s
Transitivity of $\leq$	0.001s
$1 \leq 2$	0.002s
Presburger arithmetic with bit vectors	
$x + y \geq x$	0.002s
$x_1 \leq x_2 \leq x_3 \Rightarrow x_1 + y \leq x_2 + y \leq x_3 + y$	3.651s
$x_1 \leq x_2 \wedge y_1 \leq y_2 \Rightarrow x_1 + y_1 \leq x_2 + y_2$	0.057s
$x_1 \leq x_2 \leq x_3 \wedge y_1 \leq y_2 \leq y_3 \Rightarrow x_1 + y_1 \leq x_2 + y_2 \leq x_3 + y_3$	55.190s
$1 \leq x + y \leq 5 \wedge x \geq 3 \wedge y >= 4$	12.733s
same but with iterations factorized	8.696s
Other	
automata inclusion	0.020s
$\bigvee_{i=1}^{n} P_i \wedge \bigwedge_{i=1}^{n} \neg P_i$	0.001s
$P_1 \wedge \bigwedge_{i=1}^{n} (P_i \Rightarrow P_i + 1) \wedge \neg P_{n+1} \vert n \geq 0$	0.001s

Figure 3: Provided examples and indicative execution time.