RegSTAB

Version 1.4.1

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

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

December 8, 2009

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 [ACP09b]. 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. This is all described in detail in [ACP09b].

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 (which is done in [ACP09a]). 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 a procedure very similar to RegSTAB is studied in [ACP10].

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.

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[1].
- If you're under Windows: MinGW[2].
- If you want to run tests: OUnit[3] (tested with 1.0.3) and the Findlib[4] library manager (tested with 1.2.4).

2.2 Win32

The Win32 archive contains the following:

- `QUICKSTART`: "Short manual".
- `bin/`: Contains `regstab`, `regstab.opt`, `sch2cnf`, `sch2cnf.opt`. Files with suffix `.opt` are Win32 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

2.3 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)

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

- `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 (2s vs. 30s 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[5] 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[6]. 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 vim syntax file into the directories `$HOME/.vim/syntax` and `<PREFIX>/share/apps-regstab/vim`.

[5] `http://godi.camlcity.org/godi/index.html`
[6] `http://download.camlcity.org/download/godi-rocketboost-20090421.tar.gz`

3 Usage

`regstab.opt` [-verbose] [$file$]
`regstab` [-verbose] [$file$]

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:

-verbose Be verbose, currently only displays the input formula after parsing. Allows to check that the input formula is indeed the formula understood by RegSTAB.

4 (Informal) Language Definition

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 lower bound*[7]: 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*[8] 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 $op \in \{$`<,<=,=,>=,>`$\}$

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.

[7] This can be easily circumvented **if** $n > 1$ by manually unfolding the first ranks: $\bigwedge_{i=1}^{n} S_i \wedge \bigwedge_{i=2}^{n} T_i$ is equivalent, *if* $n > 1$, to $S_1 \wedge \bigwedge_{i=2}^{n} S_i \wedge \bigwedge_{i=2}^{n} T_i$

[8] 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.

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.

5 Examples

See the directory `examples`.

6 Tools

6.1 sch2cnf

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

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

Free domain.

References

[ACP09a] Vincent Aravantinos, Ricardo Caferra, and Nicolas Peltier. A DPLL Proof Procedure For Propositional Iterated Schemata. In *Workshop Proceedings of the* 21^{st} *European Summer School in Logic, Language and Information (Worskhop Structures and Deduction)*, 2009.

[ACP09b] Vincent Aravantinos, Ricardo Caferra, and Nicolas Peltier. A Schemata Calculus For Propositional Logic. In *Proceedings of the* 18^{th} *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2009)*, LNCS. Springer, 2009.

[ACP10] Vincent Aravantinos, Ricardo Caferra, and Nicolas Peltier. Complexity of the Satisfiability Problem for a Class of Propositional Schemata. 2010. Submitted.