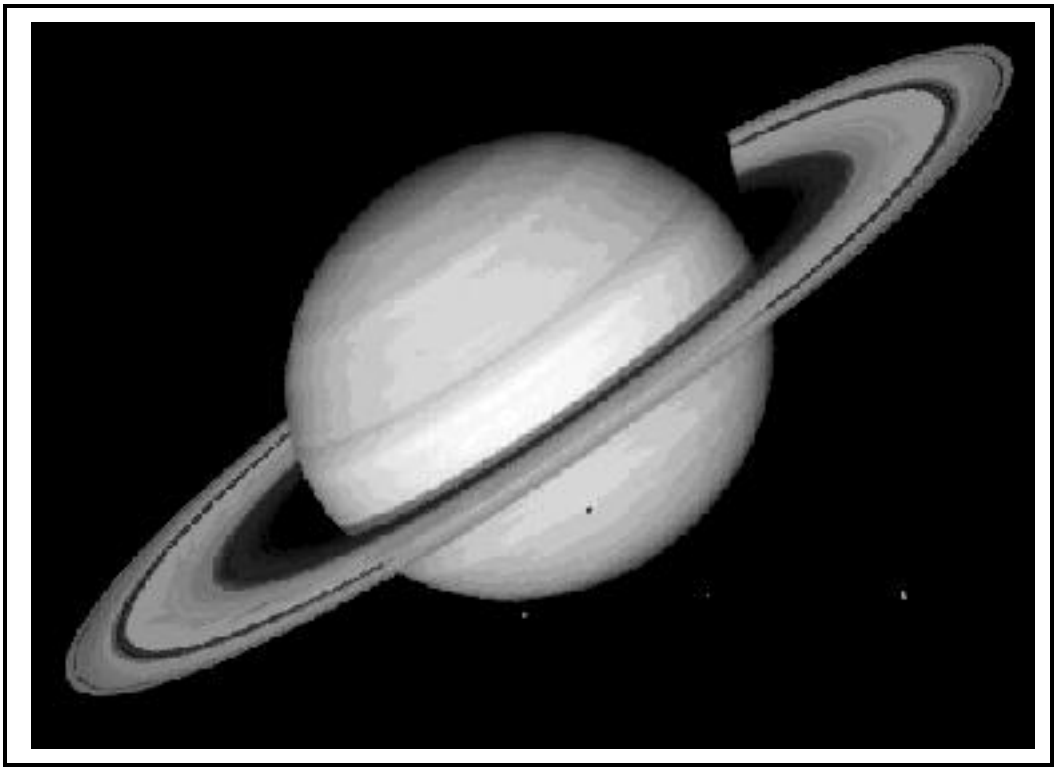
`HolSatLib` Documentation

Version 1.0β

Mike Gordon

June 7, 2002



Preface

This document describes `HolSatLib` as distributed with the *Kananaskis* release of Hol98. Section 2 describes how to install it with *Taupo* releases.

`HolSatLib` provides a very simple harness in Hol98 for invoking SAT solvers on HOL terms. Currently the following solvers are supported

Solver	**Home Page**
SATO	`http://www.cs.uiowa.edu/~hzhang/sato.html`
GRASP	`http://sat.inesc.pt/~jpms/grasp`
ZCHAFF	`http://www.ee.princeton.edu/~chaff/zchaff.html`

These solvers all require input in the standard DIMACS format[1] for conjunctive normal form (CNF). It should be straightforward to add other DIMACS compatible SAT solvers.

The purpose of `HolSatLib` is to provide a platform for experimenting with combinations of theorem proving and SAT. Hol98 can be used to deductively manipulate terms into CNF as required for SAT analysis, and then the results of the analysis can be reimported into HOL and either checked or just trusted.

Currently `HolSatLib` has only been tested under Linux, though it should be possible to run it under Windows.

Mike Gordon
June 7, 2002

[1] `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/`

Contents

1 Introduction

The following examples illustrates `HolSatLib` in action.

```
- load "HolSatLib"; open HolSatLib
[ output omitted ]
> val it = () : unit

- show_tags := true;
> val it = () : unit

- satOracle grasp ‘‘(x \/ ~y \/ z) /\ (~z \/ y)‘‘;
> val it = [oracles: grasp] [axioms: ] []
           |- z /\ y ==> (x \/ ~y \/ z) /\ (~z \/ y) : thm

- satProve grasp ‘‘(x \/ ~y \/ z) /\ (~z \/ y)‘‘;
> val it = [oracles: ] [axioms: ] []
           |- z /\ y ==> (x \/ ~y \/ z) /\ (~z \/ y) : thm
```

Setting `show_tags` to `true` makes the Hol98 top level print theorem tags.

The function `satOracle` takes a SAT solver (currently either `sato`, `grasp` or `zchaff`, but more could be added) and a term t and

1. writes a DIMACS format file corresponding to the term t
2. invokes the solver on the file to create an output file
3. parses the output file to extract the model found
4. creates a theorem, tagged with the name of the solver, that shows the model.

The function `satProve` performs steps 1–3 above, but then uses Hol98 to check that the model is really a model and then returns an untagged theorem. Note that checking a model is generally much quicker than finding it (one just ‘evaluates’ the term with the values supplied by the model).

Thus if one is prepared to trust the solver then use `satOracle`, but if one wants to verify the results (which could be time-consuming) use `satProve`.

The next example illustrates what happens on unsatisfiable terms.

```
- satOracle grasp ``(x \/ ~y \/ z) /\ ~z /\ y /\ ~x``;
> val it = [oracles: grasp] [axioms: ] []
           |- ~((x \/ ~y \/ z) /\ ~z /\ y /\ ~x)

- satProve grasp ``(x \/ ~y \/ z) /\ ~z /\ y /\ ~x``;
! Uncaught exception:
! satProveError
```

If a term t is unsatisfiable then `satOracle` will return `|- ~`t, tagged with the name of the SAT solver used. However, `satProve` will raise an exception, since there is no efficient way to check for unsatisfiability using pure Hol98 theorem proving.

A tautology checker that uses SAT can be easily programmed using `CNF_CONV`, which is supplied in the structure `canonTools` that comes with `HolSatLib`. To check the validity of a term t

- `[th1]` use `CNF_CONV` to prove `|- ~`$t = t'$, where t' is in CNF;
- `[th2]` use SAT to prove `|- ~`t';
- `[th3]` by negating both sides of `th1`, prove `|- ~~`$t = $ `~`t';
- `[th4]` hence by combining `th2` and `th3` derive `|- ~~`t.
- `[th5]` hence by the law of double negation conclude `|-` t.

Example Hol98 code to mechanise these steps is as follows:

```
(* NOT_CLAUSES = |- (!t. ~~t = t) /\ (~T = F) /\ (~F = T)  *)
val NOT_NOT = CONJUNCT1 NOT_CLAUSES;

fun SAT_TAUT_CHECK sat_solver t =
 let val th1 = canonTools.CNF_CONV(mk_neg t)
     val th2 = satOracle sat_solver (rhs(concl th1))
     val th3 = AP_TERM ``$~`` th1
     val th4 = EQ_MP (SYM th3) th2
     val th5 = EQ_MP (SPEC t NOT_NOT) th4
 in
  th5
 end;
```

2 Installing HolSatLib under Linux

1. Visit `http://www.cl.cam.ac.uk/~mjcg/HolSatLib` and download the file `HolSatLib.tar.gz`

2. place `HolSatLib.tar.gz` in a directory *dir* (where *dir* is an absolute path name)

3. connect to *dir* and execute

   ```
   gunzip HolSatLib.tar.gz; tar -xf HolSatLib.tar
   ```

   this should result in a directory *dir*`/HolSatLib` containing

   ```
   Cnf.sml HolSatLib.sig HolSatLib.sml SatSolvers.sml doc sat_solvers
   ```

4. connect to *dir*`/HolSatLib` and execute

   ```
   Holmake cleanAll; Holmake
   ```

   you should see

   ```
   Analysing HolSatLib.sml
   Trying to create directory .HOLMK for dependency files
   Analysing HolSatLib.sig
   Compiling HolSatLib.sig
   Analysing SatSolvers.sml
   Compiling SatSolvers.sml
   Compiling HolSatLib.sml
   Analysing Cnf.sml
   Compiling Cnf.sml
   ```

5. download SATO, GRASP and ZCHAFF into the directories `sato`, `grasp`, `zchaff`, respectively, in *dir*`/HolSatLib/sat_solvers` (versions may already be there)

6. after starting Hol98 execute

   ```
   loadPath := "dir/HolSatLib" :: !loadPath;
   ```

7. you should now be able to execute

   ```
   load "HolSatLib"; open HolSatLib;
   ```

3 HolSatLib Documentation

HolSatLib currently comes with three modules

Module	Description
HolSatLib	functions for invoking SAT solvers
SatSolvers	specifications of sato, grasp and zchaff
Cnf	tool for converting HOL terms to CNF (from Joe Hurd)

3.1 Contents of HolSatLib module

The signature of HolSatLib is shown below, followed by a description of the components.

```
signature HolSatLib = sig
  datatype sat_solver =
   SatSolver of {name           : string,
                 URL            : string,
                 executable     : string,
                 notime_run     : string -> string * string -> string,
                 time_run       : string -> (string * string) * int -> string,
                 only_true      : bool,
                 failure_string : string,
                 start_string   : string,
                 end_string     : string}
  val sato          : sat_solver
  val grasp         : sat_solver
  val zchaff        : sat_solver
  val tmp_name      : string ref
  val sat_command   : string ref
  val prefix        : string ref
  val showSatVarMap : unit -> int * (string * int) list
  val satOracle     : sat_solver -> Term.term -> Thm.th
  val satProve      : sat_solver -> Term.term -> Thm.thm
  val readDimacs    : string -> Term.term
```

3.1.1 sat_solver

The datatype `sat_solver` is defined in the module SatSolvers. The data in the record argument to the constructor `SatSolver` is an ad-hoc list of what is needed to invoke a SAT program and parse the results. One only needs to know what the fields contain if one is adding another SAT prover. See the source code `SatSolvers.sml` for some information in the comments.

3.1.2 sato, grasp and zchaff

The ML identifiers `sato`, `grasp` and `zchaff` are bound by module `SatSolvers` to descriptions of the corresponding SAT solvers. These descriptions are passed to `satOracle` and `satProve` to select which SAT solver to invoke.

3.1.3 tmp_name, sat_command, prefix and showSatVarMap

The reference `tmp_name` contains the temporary file name used in the last invokation of a SAT solver by `satOracle` or `satProve`. This name was generated using `FileSys.tmpName`.

The reference `sat_command` contains the actual command executed (using `Process.system`) for the last invokation of a SAT solver. This command reads from an input file and writes to an output file. The file names are generated by extending `tmp_name` (the input file name extension is `cnf` and the out extension is the name of the SAT solver used).

The reference `prefix` contains the string that is concatenated to numbers to get the HOL variables used when reading a separately generated DIMACS file with `readDimacs`. Default value is `"v"`.

The function `showSatVarMap` returns a pair consisting of the one plus the number of variables used (i.e. the first number not currently used as a variable) and the mapping from variable names to numbers for encoding a term in DIMACS format by `satOracle` or `satProve`.

3.1.4 satOracle

`satOracle` *solver term*

1. writes a DIMACS format file corresponding to *term*

(a) the mapping from HOL variable names to integers can be seen using `showSatVarMap`

(b) the input file name is *tmp*`.cnf`, where the string *tmp* is in the reference `tmp_name`

2. invokes *solver* on the file and writes results to an output file

(a) the default settings (time, verbosity etc.) supplied by *solver* are used

(b) the output file name is *tmp*`.`*name*, where the string *tmp* is in the reference `tmp_name` and *name* is the string given as the value of the field `name` of *solver*

(c) the actual command executed can be seen in the reference `sat_command`

3. parses the output file to see if a model was found and if so extracts it

(a) the presence of the string given as the value of the field `failure_string` in *solver* is assumed to indicate that *term* is unsatisfiable

(b) if *term* is not unsatisfiable, the model is assumed to be supplied as a list of integers in the output file *tmp*`.`*name* between the strings given as the values of the fields `start_string` and `end_string` in *solver*

(c) the mapping available via `showSatVarMap` is used to turn the extracted model into a HOL term

(d) if the value of the field `only_true` is `true` then it is assumed that only the positive literals of the model are given (this is the case with SATO) and so the negative literals are taken to be the negations of those variables occurring in *term*, but not in the computed model

4. creates a theorem, tagged with the name of *solver*, showing the result

(a) if *term* is unsatisfiable the result is the tagged theorem `|- ~`*term*

(b) if a model is found the result is the tagged theorem `|-` *model* `==>` *term*, where *model* is the conjunction of the literals extracted from the model

(c) the oracle tag is the name of *solver*.

3.1.5 satProve

`satProve` *solver term* goes through the same steps 1,2 and 3 as `satOracle`, but instead of step 4

1. if a model is found, then proof in Hol98 is used to first check the model is really a model (by 'evaluating' *term* using the model) and if it is an untagged theorem `|-` *model* `==>` *term* is returned
2. if a model is found by *solver*, but the Hol98 check fails, i.e. the model is invalid, then the exception `satProveError` is raised
3. if *term* is found to be unsatisfiable by *solver*, the exception `satProveError` is raised

3.1.6 readDimacs

`readDimacs` *file* reads a DIMACS format file and returns an CNF HOL term corresponding the the SAT problem in the file names *file*. The integers in the file are prefixed with the string in the reference `prefix` (the default is `"v"`).

`readDimacs` is mainly intended as a tool for getting CNF examples by reading in examples from the DIMACS problem set, which is distributed with `HolSatLib` in the directory `HolSatLib/doc/DIMACS` or is available from `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/benchmarks/cnf/`.

3.2 Contents of `SatSolvers` module

`SatSolvers` contains the definition of the datatype `sat_solver` for specifying SAT solvers.

The record that is supplied as an argument to the constructor `SatSolver` has the following fields.

`name`	name of the SAT solver
`URL`	URL of the SAT executable for downloading
`executable`	name of the SAT solver command
`notime_run`	evaluating `notime_run` *ex (infile,outfile)* returns a string giving a command to execute to run the SAT solver from input *infile* and produce output *outfile*; the paramenter *ex* should be the full path name of the SAT solver command all command options are the defaults (see solver documentation)
`time_run`	evaluating `time_run` *ex ((infile,outfile),time)* returns a string giving a command to execute to run the SAT solver for *time* units of time (the units are specified in the SAT solver's documentation) from input *infile* and produce output *outfile*; the paramenter *ex* should be the full path name of the SAT solver command; all command options, besides the time, are the defaults (currently `time_run` is not used)
`failure_string`	string whose presence in the solver output indicated unsatisfiability
`start_string`	string indicating start of model
`end_string`	string indicating end of model

Note that if a model is found, it is assumed to be bracketed by `start_string` and `end_string`. SAT solvers (like `satz`[2]) for which models are not bracketed by a fixed pair of strings cannot currently be specified for use with `HolSatLib`. If access to such solvers is needed, then it will be necessary to extend the datatype `sat_solver` to contain additional parsing data (e.g. regular expressions).

3.3 Contents of `canonTools` module

The module `canonTools` contains a simple conversion `CNF_CONV : term -> thm`, from Joe Hurd, to convert HOL terms to a form suitable for inputting to `satOracle` or `satProve`.

`CNF_CONV` t returns a theorem `|-` t `=` t', where t' is in CNF.

There are other tools for converting to various canonical forms. See the source code `canonTools.sml` for details.

[2] `http://www.laria.u-picardie.fr/~cli/EnglishPage.html`