The Kettle Manual

Version 1.10.5 of Sat Sep 25 10:48:27 PDT 2004

1 Introduction

Kettle is a theorem prover using the cooperating-decision procedures model due to Shostak and Nelson. It was originally designed as the theorem proving engine for constructing proof-carrying code. As such, Kettle can produce concrete representations of proofs for all facts that it proves.

This documentation is very incomplete at the moment.

A PDF version and a PS version of this document.

online API documentation

2 Installation

Kettle should run on any architecture that has an Ocaml compiler. We use it all the time on Linux and on Windows/cygwin.

If you are using Windows, get the cygwin distribution and make sure to get all Development tools (Dev), the XFree86-bin, XFree86-lib and XFree86-prog packages (XFree86) as well as TclTk package (lib). (you need the latter to use the the GUI). You should get also the ocaml package.

Next you need to tell Ocaml where Tcl is located: add an environment variable

```
export TCL_LIBRARY=/usr/share/tcl8.4
```

On cygwin, you can alternatively set in the Control Panel/System/Advanced `TCL_LIBRARY` to `c:\cygwin\usr\share\tcl8`

Now you can configure and build Ocaml (if not on cygwin):

```
./configure
# Make sure the last step has configured Labltk
make world
make opt
make install
```

Now you are ready to build Kettle:

1. Download the Kettle distribution.
2. Unzip and untar the source distribution. This will create a directory called `kettle` whose structure is explained below.
   `tar xvfz kettle-1.10.5.tar.gz`
3. Enter the `kettle` directory and run the `configure` script and then GNU make to build the distribution. If you are on Windows, at least the `configure` step must be run from within `bash`.

```
cd kettle
./configure
make
cd test
make quicktest
```

This should leave the executable file obj/x86_WIN32/kettle.asm.exe (or x86_LINUX as might be the case).

To test that you have Kettle working you can run:

```
cd test
make quicktest
```

To test if the (experimental) GUI works, do

```
cd test
make prove/equality/uninterpreted GUI=1
```

The above command will run the prover on the file `equality/uninterpreted.cvc` and will leave a proof in the file `equality/uninterpreted.prf`.

3 The Kettle Theorem Prover

3.1 The Input Language

Kettle supports goals written either in the CVS syntax, or in the LF syntax. In lieu of a documentation, you can start by looking at examples in the `test/uninterpreted` directory. You can find examples of LF inputs in the files with extension `.p`, and examples of CVC input are in files with the extension `.cvc`.

3.2 Command Line Options

Kettle must be invoked with the names of the files containg goals to prove. Additionally, Kettle supports the following command line options:

- `-help` Print a help message
- `-lfinput` Use LF input language (default)
- `-cvsinput` Use the CVS input language
- `-verbose` Turn on verbose mode
- `-log xxx` Sends the logging output to the given file. By default this output is sent to `stderr`
- `-o xxx` Write the proof in the given file.
- `-listprocs` List the identifiers of the satisfiability procedures that have been built in Kettle.
- `-enable XX` Enable the given satisfiability procedure. Multiple `-enable` options may be given. Use `-enable ALL` to enable all procedures.
- `-disable XX` Disable the given satisfiability procedure. Multiple `-disable` options may be given. Use `-disable ALL` to disable all procedures.
- `-debug XX` Turn on debugging output for the given satisfiability procedure.
- `-lfi` Use LFi representation for proofs.
- `-oracle` Use the oracle representation for proofs. At the moment this works only for select procedures.
- `-checkproofs` Do proof checking before printing the proofs. This is useful for discovering prover bugs.
- `-nostop` Does not stop on errors.
- `-pause` Gives you a chance to continue after an error.
- `-gui` Use the Graphical User Interface.

4 The Graphical User Interface

If you invoke Kettle with the `-gui` argument, it will invoke the GUI **after** proving a fact. The purpose of the GUI is to show what goals were successful (and to show a proof for them) and which were not. The GUI is only useful if the input to Kettle contains annotations naming the various parts of the predicate to be proved. We discuss below an example of an input file containing GUI annotations, but if you are anxious you can find this example in the `test/arith/gui.p` file and you can play with it by running:

```
> cd test
> make prove/arith/gui EXTRAARGS=-gui
```

4.1 The GUI Annotations

For the GUI to work the input file must contain annotations in the form of a new predicate constructor. For example, if p is a predicate then the following is an annotated version of p, saying that the name of the predicate is `"this is p"` and that it corresponds to line 9 in the file named `foo.java`:

```
(block [file:"foo.java",line:9,name:"this is p"]  p)
```

This section is written using the LF input language. For the CVC input language the same is true except that the syntax for the above example is

```
BLOCK [file:"foo.java",line:9,name:"this is p"]
p
ENDBLOCK
```

There must be a block annotation surrounding each top-level predicate to be proved. This will give the name that the GUI will show for the top-level predicate.

Each assumption and each goal that is encountered in the predicate are associated with the innermost block that contains them. If you want the maximum amount of information in the GUI you must have a block annotation for each assumption and for each goal. For example, the following input describes a predicate with two parts, the first part having an assumption and two goals.

```
Sat_p1 : pf
  (block [file:"foo.java",line:9,name:"this is p"]
    (block [file:"foo.java",line:9,name:"Part1"]
       (=> (x < 0)
           (block [file:"foo.java",line:10,name:"Part1-Goal1"]
              (<= (+ x 1) 0)
           )
           (block [file:"foo.java",line:11,name:"Part1-Goal2"]
              (<= (+ x 1) 0)
           ))
    )
    (block [file:"foo.java",line:29,name:"Part2"]
           (block [file:"foo.java",line:11,name:"Part2-Goal1"]
              (= 0 0)
           )
    )
  )
```

The nested structure of the blocks gives the hierarchy.

4.2 The GUI panes

The GUI has four panes: Position, Source, Goals and Assumptions. The Position pane shows a hierarchical view of the verification steps. There is a root node, whose children are the individual predicates to be proved. Nodes with a little triangle handle are internal nodes in the hierarchy. If you click on the handle you can collapse or expand the hierarchy under that node. If you click on the label for a node, then the source pane shows the line to which it corresponds.

The Assumptions pane shows the assumptions that are in scope at the start of a block (not including the assumptions in the block itself). Typically, as you go does in the hierarchy, you will have more assumptions. Assumptions have names (e.g. `H_1`), which are refered from the proofs.

The Goal pane shows different things for different kinds of nodes. If you are on a goal node (typically a leaf) then you will see the goals for that node, along with their proofs. If you are on an assumption node, you will see in the Goals pane the assumption(s) that the node is adding.

In both the Assumptions and the Goals panes you can double click on a line and you get a pretty-printed version of the line in a separate window.

If all goals are successful then the GUI will start with all nodes collapsed. If there are failed goals then the tree is expanded to show all the failures. Also, the failed goals and all the parent nodes are collored red.

The GUI will try to find the files to show in the Source pane based on the information contained in the predicate annotations, along with a set of search directories (semicolon separated path names). The search directories, along with all configuration elements of the GUI are saved in a file called `kettle_gui.cfg` in the current directory and are restored automatically at the next invocation.

5 The Kettle Architecture

Kettle can parse input files in the LF or CVC format.

The code for the main proving engine is in Engine. The engine is given a thunk that when invoked will return the next input fragment. Input fragments are things like a literal to prove in the current context, an assumption to add to the current context, an indication to pop the last assumption, etc. The data type for the input language is described in the Input file. This architecture was essential for proving very large sequences of literals, what would not all fit in memory at once.

The actual inputs are written in terms of logical expressions and formulas, defined in Logic. Proofs of formulas can be written using the language defined in Proof. The same module also defines a proof checker that is enabled when you pass the `--checkproofs` argument on the command line.

The Engine module is initialized with a list of satisfiability procedures to use, and a proof generator. A satisfiability procedure consists of one or more ML modules, one of which exports a structure of type Satproc.entries. That structure contains entry points to the main operations that a satisfiability procedure supports.

In order to add a satisfiability procedure you can take the following steps:

- Choose a short name for it (say, “foo”). Then change the `configure.ac` file to add to the list of “features” the name of the procedure. Add “foo=yes”, if you want it to be enabled by default, or “foo=no” if you want it disabled by default. Make sure to run “autoconf” when you edit this file.
- Create the ML files that implement the procedures. Say that you have the files “src/foo1.ml”, “src/foodir/foo2.ml”. Say that the module foo1 contains the entry point.
- In Makefile.in add a section

```
ifdef USE_FOO
SATMODULES += foo1 foo2
SOURCEDIRS += src/foodir
endif
```

- Further down in Makefile.in, where the file features.ml is being described, add

```
ifdef USE_FOO
   echo ‘‘   Foo1.entries; ‘‘ >> $@
endif
```

- Now you can include the module foo in the build (if not included by default), by doing `configure --with-foo`.
- If the module is included in the build, you can use the command line options `--enable` and `--disable` to enable or disable it for a particular run.

Also, before starting the engine, you must set the current UI to use. If you pass the `--gui` argument to Kettle, you will use the GUI, otherwise you use a text user interface.

6 Annotation Language

The following is the grammar for annotations:

<table>
<tr><td>Annotation</td><td>A</td><td>::=</td><td>ANN(AName, AA$_1$,AA$_2$, . . .)</td></tr>
<tr><td>Annotation name</td><td>AName</td><td>::=</td><td>Predident | Ident</td></tr>
<tr><td>Annotation argument</td><td>AA</td><td>::=</td><td>E | P | "..." | { Reg$_1$,Reg$_2$, . . . }</td></tr>
<tr><td>Predicate start</td><td>startpred</td><td>::=</td><td>’A’ - ’Z’ | ’¡’ | ’¿’ | ’=’</td></tr>
<tr><td>Expression start</td><td>startident</td><td>::=</td><td>’a’ - ’z’ | ’_’ | ’%’ | ’$’ | ’!’ | ’ˆ’ | ’&’ | ’*’
| ’+’ | ’-’ | ’\’ | ’?’ | ’—’ | ’/’ | ’ ’</td></tr>
<tr><td>Predicate constructor</td><td>Predident</td><td>::=</td><td>Startpred (Startpred | Startident | [’0’ - ’9’]) *</td></tr>
<tr><td>Expression constructor</td><td>Ident</td><td>::=</td><td>Startident (Startpred | Startident | [’0’ - ’9’]) *</td></tr>
<tr><td>Number</td><td>Num</td><td>::=</td><td>[’0’ - ’9’] +</td></tr>
<tr><td>Register</td><td>Reg</td><td>::=</td><td>Architecture dependent gcc notation</td></tr>
<tr><td>Predicate</td><td>P</td><td>::=</td><td>TRUE | FALSE
| (AND P$_1$ P$_2$ . . .) | (OR P$_1$ P$_2$ . . .)
| (=> P$_1$ P$_2$) | (ALL [Ident] P) | (EXISTS [Ident] P)
| (Predident E$_1$ E$_2$ . . .)</td></tr>
<tr><td>Expression</td><td>E</td><td>::=</td><td>Num | Reg | Ident
| (+ E$_1$ E$_2$) | (- E$_1$ E$_2$) | (* E$_1$ E$_2$) | Num(E) | (Ident E$_1$ E$_2$ ..</td></tr>
</table>

Important syntactic notes:

- Identifiers are classified in two separate categories, depending on the first letter. If the first letter is a capital letter or one of the characters “>”, “<”, or “=” then the identifier is a predicate constructors. Otherwise it is an expression constructor, or a variable.
- Register names typically have the same syntax as expression constructors, but the parser will recognize them by their standard `gcc` notation. For example, on MIPS “$a0” and “$4” are two names for the 4th register.
- The names for the logical connectives are all capitals and are case sensitive. If you write “true” it will be taken as the expression constructor “true”. (Maybe we ought to declare the arity of new constructors in order to catch simple typos like this?)
- The “n(e)” expression is an abbreviation for “(sel mem (add n e))”.

7 Changes