Victor: a SPARK VC Translator and Prover Driver

User Manual for release 0.9.0

Paul Jackson
`pbj@inf.ed.ac.uk`

5th November 2010

Contents

1 Supported provers and prover languages

Victor has API interfaces to the Cvc3 and Yices provers, and can drive any prover that accepts Simplify or SMT-LIB format input files.

1.1 Simplify language

The Simplify language is supported by the Simplify and Z3 provers.

1.2 SMT-LIB language

The SMT-LIB initiative (http://www.smtlib.org) defines a standard language for formatting input to SMT solvers and collects benchmarks in this format.

Provers taking SMT-LIB input must support at least one of the SMT-LIB sub-logics

- AUFLIA: quantifier formulas involving arrays, uninterpreted functions, linear integer arithmetic.
- AUFLIRA: quantifier formulas involving arrays, uninterpreted functions, linear integer and linear real arithmetic.
- AUFNIRA: quantifier formulas involving arrays, uninterpreted functions, non-linear integer and non-linear real arithmetic.
- UFNIA: Non-linear integer arithmetic with uninterpreted sort, function, and predicate symbols.

Such provers include Alt-Ergo, CVC3, Yices and Z3. Victor currently makes no use of the support for arrays or the reals.

1.3 Alt-Ergo

Alt-Ergo is an open-source SMT solver from LRI (Laboratoire de Recherche en Informatique) at Université Paris-Sud. It is available from http://alt-ergo.lri.fr/.

Victor has been tested most recently with the V0.92.2 release using Alt-Ergo's SMT-LIB file-level interface.

If the standalone Alt-Ergo executable is downloaded rather than built from the Alt-Ergo source distribution, it is also necessary to copy the file `smt_prelude.mlw` from the source distribution to the `run/` directory.

1.4 CVC3

CVC3 is an open-source SMT solver jointly developed at New York University and the University of Iowa. It is available from http://www.cs.nyu.edu/acsys/cvc3/.

Victor can link to a CVC3 library and can then drive CVC3 via its API. Alternatively Victor can invoke a CVC3 stand-alone executable on SMT-LIB format files.

Victor has been tested with the latest release, V2.2, dating from November 2009. CVC3 is significantly slower than Yices or Z3 (maybe 5-10$\times$), especially when VCs are unprovable. It has some basic support for non-linear arithmetic.

When driven via its API, this version of CVC3 throws exceptions and has some segmentation faults on a few of the SPARK VCs from the tokeneer set, The exceptions are caught and reported by Victor, but the segmentation faults cause Victor to halt. To enable Victor runs in the presence of these faulting problems, it is possible to tell Victor to ignore trying to run CVC3 on certain VCs.

1.5 Simplify

Simplify is a legacy prover, used most notably in the ESC/Java tool.

The Modula-3 sources and some documentation are available from HP labs. Visit `http://www.hpl.hp.com/downloads/crl/jtk/index.html` and follow the "Download Simplify here" link.

Executables for Linux and other platforms can be pulled out of the ESC/Java2 distribution: visit `http://secure.ucd.ie/products/opensource/ESCJava2/`. In October 2007, the executables for V1.5.4 were found in a file `Simplify-1.5.5-13-06-07-binary.zip`.

Simplify has good performance, but is unsound and sometimes crashes because it uses fixed-precision integer arithmetic.

Victor interfaces to Simplify using temporary files and by invoking the Simplify executable in a sub-process. Unlike the case with CVC3, Victor can tolerate Simplify crashing. Victor provides notifications of Simplify crashes in its output files.

1.6 Yices

Yices is a state-of-the-art SMT solver available from SRI at `http://yices.csl.sri.com/`.

Victor links with a Yices library provided with the Yices distribution. Victor has been tested with the latest public release, V1.0.24. This version, bug fixes apart, dates from summer 2007 and essentially is the version that lead the field in the 2007 SMT competition.

Yices is fussy about VCs containing non-linear arithmetic expressions. Victor currently just has Yices ignore any hypotheses or conclusions containing such expressions, and, not infrequently, VCs are provable despite these ignored VC clauses.

Yices will accept universally-quantified hypotheses with non-linear arithmetic expressions, and sometimes can make use of linear instantiations of these. Unfortunately, the current behaviour on finding a non-linear instantiation is abandon the proof attempt rather than simply ignore the instantiation.

No crashes have been observed with recent versions of Yices. However, on a few VCs (not in the test sets provided with the distribution), Yices just keeps going on and on. No mechanism for timing out on such cases has yet been implemented, the only way to deal with them is to request that Victor ignore them.

Victor can also drive Yices using SMT-LIB format files.

1.7 Z3

Z3 is a state-of-the-art SMT solver developed at Microsoft. See `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.

Victor has been tested most recently with the Linux version of release 2.13. No problems have been observed with this version.

Z3 has good performance and better VC coverage than other solvers tried. In particular, it has the best support for non-linear arithmetic.

Victor interfaces to Z3 using temporary files and by invoking the Z3 executable in a sub-process. The temporary files can be in either SMT-LIB or Simplify format.

2 Installation and Testing

Victor is written in C++ and currently only runs on Linux. The current distribution includes some preliminary code to allow it to compile and run on Windows. However, this code has not yet been fully tested.

At Edinburgh, Victor is curently compiled and run on a Red Hat Fedora 13 machine. It makes use the following tools:

- `make` V3.81
- `gcc/g++` V4.4.4
- `bison` V2.4.1
- `flex` V2.5.35

The main external library it uses is

- `gmp` V4.3.1

Its precise dependencies on these versions are largely unknown. One observation is that some tweaks to the `bison code` in `parser.yy` were necessary when shifting from `bison` V2.3 to `bison` V2.4. Comments in `parser.yy` indicate what needs to be changed for compilation with V2.3

By default, the `gmp` library is dynamically linked in. If running a single executable on several different Linux platforms, this can cause problems and it might be desirable to use static linking instead. To achieve this, use `STATIC_GMP=true` on the `make` command line when building Victor.

To install:

1. Untar the distribution. E.g.

   ```
   tar xzf vct-0.9.0.tgz
   ```

   This should generate a top level directory `vct-0.9.0` including subdirectories `src`, `bin`, `run`, `vc` and `doc`. The `doc` directory includes a copy of this manual. Other directories are described below.

2. Configure Victor for each of the provers you wish to use it with.

   Cvc3: To enable the API driver, uncomment the definition of variable `CVC3DIR` in file `src/Makefile` and edit its value to be that of your CVC3 installation.

   To use the SMT-LIB format file interface, ensure that an executable `cvc3` is on your current path.

   Simplify: Ensure an executable called `simplify` is on your current path.

   Yices: To enable the API driver, uncomment the definition of variable `YICESDIR` in file `src/Makefile`, and edit its value to be that of your Yices installation.

   To use the SMT-LIB format file interface, ensure that an executable `yices` is on your current path.

   Z3: Ensure an executable called `z3` is on your current path.

   Alt-Ergo: Ensure an executable called `alt-ergo` is on your current path.

   Alternate names, and optional paths can be specified for each executable at the top of the `Makefile` in the `run` directory.

   Victor can be run without driving any prover. This is useful for testing if Victor's parser can handle certain VCs and for gathering information on VCs. This mode can be used for compiling reports on the coverage obtained with the Simplifier prover provided with Praxis's SPARK toolkit.

3. Build a Victor executable by `cd`ing to the `src` directory and typing `make`. This does a variety of things, including

   (a) Creating `.d` files recording `make` rules that capture dependencies between source files.

   (b) Running the bison parser generator and the flex lexer generator.
   (c) Compiling various .o files.
   (d) Linking the .o files together, along with prover libraries and the gmp library, and installing the resulting executable named vct in the bin directory.

   For convenience, a sub-directory build contains copies of files created during a build of Victor where it was configured for running with the Simplify and Z3 provers. For example, if you do not have the correct version of bison, you could copy over the bison output files to the src directory.

4. Add the vct/bin directory to your PATH.
5. Build utility tools for analysing the VCT and VLG comma-separated-value output files created by Victor. Enter make csvutils. This causes the executables csvproj, csvfilt, csvmerge and csvisect to be added to the bin directory.
6. Try running Victor on the example VCs provided with the distribution. Check the output files match the provided output files. See Section 3.5 for details.

3 Operation

3.1 Terminology

We refer to SPARK VCs as *goals* and use the term *goal slice* to refer to a proof obligation build from a VC by considering just one of the goal's conclusions and ignoring the others.

A *unit name* is the hierarchical name of a program unit. The unit name with a .fdl, .rls, .vcg or .siv suffix gives a pathname for the corresponding VC file relative to the root directory of all the VC files for a SPARK program.

3.2 Basic operation

The basic operation of Victor is to

1. Read in a list of names of SPARK program units.
2. Read in the VCs described in the .fdl, .rls, .vcg file triples output by the SPARK Examiner for the named program units.[1]
3. Invoke a prover on each goal or goal slice.
4. Output .vct, .vsm and .vlg report files.

[1] Optionally it can read in the simplified .siv files output by the SPARK Simplifier instead of the .vcg files.

3.3 Input and output files

The Victor-specific input and output files are as follows.

3.3.1 Unit listing input file

Typically Victor is run on many program units at once. An input *unit listing* `.lis` file is used to indicate the units it should consider. The grammar for each line in a unit listing file is given by

line	::=	*unitname* {*option*}
option	::=	[*tag*?]*val*
val	::=	*goal*
	|	*goal*.*concl*
	|	*filename*`.fdl`
	|	*filename*`.rul`
	|	*filename*`.rlu`

where square braces ([]) enclose optional non-terminals, curly braces ({}) enclose non-terminals repeated 0 or more times, the terminals *unitname*, *tag* and *filename* are alphanumeric strings, and the terminals *goal* and *concl* are natural numbers. The meaning of the components of a line are as follows.

- *unitname* is the hierarchical name of a unit (SPARK subprogram).
- *tag* tags an option. A tagged option is only active if the tag is also supplied as one of the values of the `-active-unit-tags` Victor command-line option. Untagged options are always active.
- *goal* and *goal*.*concl* select particular goals and goal slices in the unit. The Victor command-line options `-include-selected-goals` and `-exclude-selected-goals` control how Victor treats these selected goals and goal slices.
- *filename*`.rul` and *filename*`.rlu` are auxiliary rules files to load
- *filename*`.fdl` is an auxiliary declarations file. For example, this can declare constants and functions introduced in an auxiliary rules file.

Comment lines are allowed: these are indicated by a `#` character in the first column. Also blank lines are allowed.

One way to prepare a unit listing file is to run the command

```
find . -name ’*.fdl’ | sed -r ’s/\.\/|\.fdl//g’ > units.lis
```

in the root directory of a set of VC files.

3.3.2 VCT output file

The VCT file includes one line for each goal or goal slice. The fields of each line are:

1. Path to unit. This describes the containing packages
2. Unit name, without a prefix for the containing packages.
3. Unit kind. One of `procedure`, `function` or `task_type`.
4. Source of path in subprogram for VC
5. Destination of path for VC, or VC kind if not path related
6. VC goal number
7. Conclusion (goal slice) number
8. Status (one of true, unproven, error)
9. Proof time (in sec)
10. Brief remarks about goal and solver interactions
11. Operator kinds occurring in hypotheses
12. Operator kinds occurring in conclusion

3.3.3 VSM output file

The run summary file has extension `.vsm`. It is a 1 line comma-separated-value file with the fields

1. Report file name.
2. Number of `ERROR` messages in log file
3. Number of `WARNING` messages in log file
4. Total number of goal/goal slices processed.
5. Number of goal/goal slices with *true* status
6. Number of goal/goal slices with *unproven* status
7. Number of unproven goal/goal slices that involved a timeout.

8. Number of goal/goal slices with *error* status
9. Percent of goal/goal slices with *true* status
10. Percent of goal/goal slices with *unproven* status
11. Percent of unproven goal/goal slices that involved a timeout.
12. Percent of goal/goal slices with *error* status
13. Total execution time

A file `vsm-file-header.txt` provides a 1 line comma-separated list of headings for these summary files.

Summary files can be concatenated together with a header file and then viewed in any spreadsheet program.

3.3.4 VLG output file

The VLG log file includes

1. a record of the command line options passed to Victor,
2. various information, warning and error messages,
3. statistics on the run, including numbers of VCs proven and unproven, and time taken.

3.4 Invocation of Victor

The command line syntax for invoking Victor is

`vct` [*options*] [*unitname*]

The *unitname* argument is used to identify a single unit on which to run Victor.

To run Victor on multiple units, omit the `unitname` argument and use instead the `units` option to specify a unit listing input file.

If both a *unitname* and a `units` option are provided, the `units` option is ignored.

Victor takes numerous options, many of which are currently necessary. See the next section for a description of a `Makefile` that provides standard option sets.

3.5 Examples

The `vc` directory has subdirectories for some example sets of VCs.

See the file `vc/README.txt` for further information on these sets.

The `run` directory provides a Makefile with rules for running Victor on the VC sets in the `vc` directory. These rules use Make patterns in their targets, and can easily also be used for running Victor on users' own VC sets. The rules set appropriate Victor command-line options and so allow starting Victor users to ignore having to figure these out for themselves. See `run/Makefile` for details.

Reference report files obtained from running `make` on some of these targets are included in directory `run/out-ref`. Unix `diff` can be used to check that newly-generated report files are the same as the reference files. If the command line option `-gstime` is used to include times of prover runs in report files, it will be necessary to use the `csvproj` utility to remove the field for these times in order to get files that are expected to be identical.

3.6 Performance tips

1. When SMT solvers cannot prove a goal, they often keep trying almost indefinitely rather than halting, so it is good to run them with some kind of time-out. When several VCs cannot be proven, Victor's total run-time can be dominated by the runs that go to time-out. Setting a shorter time-out can therefore sometimes radically reduce Victor's run-time, often with little or no drop in number of goals proven.
2. SMT solver performance on goals they can prove is often dependent on the number of quantified axioms. By default, Victor uses a number of quantified axioms from the rules files `divmod.rul` and `prelude.rul` in the `run/` directory. In some cases, not all these axioms are necessary, and faster run-times are achievable with alternate rules files that prune down these axiom sets.

4 Command line options

Options are specified with syntax `-`*`name`* or `-`*`name`*`=`*`value`*. Option values can be boolean (`true` or `false`), natural numbers (e.g. `42`) or strings. An option `-`*`name`* is interpreted the same as An option `-`*`name`*`=true`. An unset boolean option is interpreted as `-`*`name`*`=false`.

If the same option is given multiple times with different values, the usual behaviour is that the last value is taken. Occasionally all multiple values are used. These cases are always explicitly pointed out below.

A later option `-name=`, `-name=false`, `-name=none` or `-name=default` Clears all earlier values given for the option and makes it unset.

4.1 Input options

These options control where VCs are read from, provision of auxiliary declarations and rules, and filtering of VCs before invoking the selected prover.

`-units=`*unit-listing*
: Run on units named in *unit-listing* file.

`-prefix=`*prefix*
: Use *prefix* as a common prefix for all unit names. *prefix*`/`*unitname* should give an absolute or relative pathname for the VC file set of a program unit. Default is that no prefix is used.

`-decls=`*declfile*
: For every program unit, read auxiliary `.fdl` declarations file named in *declfile*. Multiple files can be specified using multiple

`-decls`
: options.

`-rules=`*rulesfile*
: For every program unit, read in auxiliary rules file named in *rulesfile*. Multiple files can be specified using multiple

`-rules`
: options.

`-siv`
: Read in `.siv` simplified VC files output by the Simplifier rather than `.vcg` Examiner VC files.

`-goal=`*g*
: Only consider goal number *g*. This option is intended for use when Victor is run on a single unit, when a *unit-name* argument and no `units` option is given.

`-concl=`*c*
: Only consider conclusions (goal slices) numbered *c*. This option is intended for use when Victor is run on a single unit.

`-skip-concls`
: Do not pass conclusion formulae to the selected prover. This option is good for helping to identify goals or goal slices true because of an inconsistency in the hypotheses or rules.

`-skip-hyps`
: Do not pass hypothesis formulae to the selected prover. This option is good for helping to identify goals or goal slices true because of an inconsistency in the rules.

`-from-unit=`*unit-name*
: Only drive to selected prover the units listed in the `-units` option starting with *unit-name*. Default is to start with first.

`-from-goal=`*g*
: In first unit to be passed to prover, start driving goals / goal slices to prover at goal g.

`-to-unit=`*unit-name*
: Stop driving units to the selected prover after *unit-name* is encountered. Default is to continue until the last listed unit.

`-active-unit-tags=`*tags*
: Identify which tagged options (if any) in the unit listing file to make active. See Section 3.3.1 for more on this. Multiple tags should be separated by colons (`:`).

`-include-selected-goals`
: When particular goals or goal slices are selected for a unit in the unit listing file, run Victor on just those goals or goal slices.

`-exclude-selected-goals`
: When particular goals or goal slices are selected for a unit in the unit listing file, do not run Victor on those goals or goal slices.

4.2 Translation options

See Section 5 for a presentation of these options, since they make best sense in a discussion of the overall translation process.

4.3 Prover and prover interface selection

`-prover=`*prover*
: Select the prover to drive. Valid values of *prover* are:

- `cvc3`
- `simplify`
- `yices`
- `z3`

A value of `none` can also be specified. This is useful if one just wants to generate prover input files.

`-prover-command=`*prover-command*
: Use instead of the `prover` option to specify explicitly a shell-level command for invoking the prover. This allows alternate provers or custom prover options to be specified.

Selecting neither this option or the `prover` option is equivalent to setting the value of `prover` to `none`.

`-interface-mode=`*mode*
: Select the prover interface mode. Valid values of *mode* are:

- `api`: Use prover API. Acceptable with `cvc3` or `yices` value for `prover`.
- `smtlib`: Use SMT-LIB-format files and stand-alone prover executable. Acceptable with `cvc3`, `yices` or `z3` value for `prover`, and with `prover-command` option.
- `simplify`: Use Simplify-format files and stand-alone prover executable. Acceptable with `simplify` or `z3` value for `prover`, and with `prover-command` option.
- `dummy`: Use some default code that mostly does nothing. In this case, Victor still parses the VC files, does a prover independent translation of the goals, and generates VCT and log output files. This is the default option.

4.4 Prover driving options

`-fuse-concls`
: Pass one goal at a time to the selected prover. By default Victor passes one goal slice at a time.

`-working-dir=`*working-dir*
: Use *working-dir* as root of directory tree of files used for prover input and output. An argument of '`.`' is acceptable to indicate the current directory. Defaults to `/tmp`.

Unless one of the next three options is used, the same file names are used for every every prover run and every Victor run.

`-hier-working-files`
: Use distinct files for each prover invocation and arrange in a hierarchical tree under *working-dir*.

`-flat-working-files`
: Use distinct files for each prover invocation and arrange all as members of *working-dir*.

`-unique-working-files`
: Within a given Victor run, use the same file names for each prover invocation, but, by including hostname and process number in file names, make the names unique to the Victor run. This option is useful if one wants to have simultaneous Victor runs.

`-ulimit-timeout=`*time*
: If using either of the file-level interface modes, use the Linux *ulimit* process limit facility to time out prover invocations after *time* seconds. The `time` value should be a natural number. The default is not to time out prover invocations.

`-shell-timeout=`*time*
: If using either of the file-level interface modes, use the provided shell script `timeout.sh` to time out prover invocations after *time* seconds. The `time` value can be an integer or a fixed-point number (e.g. 0.1). The default is not to time out prover invocations.

  Currently this option is not that robust and use of `-ulimit-timeout` is recommended instead.

`-logic=`*logic*
: If using the SMT-LIB interface mode, set the value of the `:logic` attribute in the SMT-LIB-format files to *logic*. The default is `AUFLIA`.

`-smtlib-hyps-as-assums`
: If using the SMT-LIB interface mode, insert each hypothesis into the SMT-LIB-format file as the value of a distinct `:assumption` attribute.

`-drive-goal-repeats=`*count*
: Repeat each prover invocation *count* times. This is used to increase precision of prover runtime measurements when using an API interface.

`-check-goal-repeats=`*count*
: Repeat each prover invocation *count* times. This is used to increase precision of prover runtime measurements when using a file-level interface.

4.5 Output options

4.5.1 Screen output options

`-utick`
: Print to standard output a `*` character at the start of processing each unit. If `-longtick` also selected, print instead the unit name.

`-gtick`
: Print to standard output a `;` character at the start of processing each goal. If `-longtick` also selected, print instead the goal number.

`-ctick`
: Print to standard output a `.` character at the start of processing each conclusion of a goal. If `-longtick` also selected, print also the conclusion number.

`-longtick`
: See above.

`-echo-final-stats`
: Print to standard output the final statistics that are included at the end of the report file.

4.5.2 General report file options

`-report=`*report-file*

Use *report-file* as body of filenames for VCT, VSM and VLG report files. Default is to use `report`.

`-report-dir=`*dir*

Put report files in directory *dir*. If directory does not exist, it is created. Default is to use current directory.

4.5.3 VCT file options

`-count-trivial-goals`

Write an entry to the VCT file for each input goal of form `*** true`. In VCG files, these are the goals proven by the Examiner. In SIV files, these are the goals proven by the Examiner or the Simplifier. These entries have status `true` in field 8 and the comment `trivial goal` in field 10.

Use this option along with option `-fuse-concls` to have the goal counts match those from the POGS (Proof Obligation Summariser) tool.

`-hkinds`

Report list of hypothesis kinds in field 11 of VCT file

`-ckinds`

Report list of concl kinds in field 12 of VCT file

`-gstime`

Report time taken by prover to process a goal slice or goal in field 9 of VCT file.

`-gstime-inc-setup`

Include setup time in gstime. This setup time is time to send declarations, rules, hypotheses and conclusions to the prover before invoking prover itself.

It is appropriate to include this time when calling the prover via an API (Yicesand CVC3 cases) since the provers do incremental processing on receiving this information. When the prover interface is via files, this setup time is the time to write an input file for the prover, so it is not as appropriate to include it.

`-csv-reports-include-goal-origins`

Include information on goal origins in fields 4 and 5 of VCT file. Default is not to include this information.

`-csv-reports-include-unit-kind`

Include information on unit kind in fields 3 of VCT file. Default is not to include this information.

4.5.4 Log file options

`-level=`*level*

Report all messages at or above priority *level*. The levels and associated names are

6	`error`
5	`warning`
4	`info`
3	`fine`
2	`finer`
1	`finest`

The *level* value can either be a number or the associated name. The default level is *warning*.

4.6 Debugging options

`-scantrace`

Write lexer debugging information to standard output

`-parsertrace`

Write parser debugging information to standard output

4.7 CVC3 options

Unless otherwise specified, these options are only relevant when invoking Cvc3 via its API. The main options are as follows.

`-counterex`

Report counterexamples for false and unknown queries.

! *I have not figured out yet how to direct* Cvc*3 to write counter-examples to files. A work-around to view counter-examples is to run with this option and the* ***`-cvc-inputlog`*** *option, and then run the standalone* Cvc*3 executable on the generated* Cvc*3 input file.*

`-timeout=`*time*

Set a timeout period in units of 0.1 seconds for runs of Cvc3, both via API and via executable. Uses Cvc3's internal support for timing out.

`-cvc-loginput`
: Enable echoing of API calls for each Cvc3 run to a file. Use the `-working-dir` option to set where the file is stored and the `-hier-working-files` and `-flat-working-files` options to control whether and how distinct files are used for each run. Files have suffix `.cvc`. If distinct files are not requested, all runs will be echoed to a file named `cvc3.cvc`, each run overwriting the previous one. These files are saved in Cvc3's standard input language and can be used as input to a Cvc3 stand-alone executable.

See the file `cvc-driver.cc` for further available options. Not all of these have been tried out yet.

4.8 Simplify options

No options are currently available.

4.9 Yices options

Unless otherwise specified, these options are only relevant when invoking Yices via its API.

`-yices-loginput`
: Enable echoing of API calls for each Yices run to a file. Use the `-working-dir` option to set where the file is stored and the `-hier-working-files` and `-flat-working-files` options to control whether and how distinct files are used for each run. Files always have suffix `.yices`. If distinct files are not requested, all runs will be echoed to the file `yices.yices`, each run overwriting the previous one. These files are saved in Yices's standard input language and can be used as input to a Yices stand-alone executable.

  ! *Victor lets Yices reject non-linear parts of formulae - see the warnings in Victor's log file. These formulae might have to be removed by hand for Yices to load these input files properly.*

`-yices-logoutput`
: Set file for output of each run of Yices. Location of file and whether distinct files generated for each run are specified in same way as with `-yices-loginput`. Suffix of files is `.ylog`. If distinct files not requested, all runs written to `yices.ylog`.

`-counterex`
: Enable reporting of counter-example models to output log file.

`-timeout=`*time*
: Set a timeout period in seconds for runs of the Yices executable on SMT-LIB-format input files. Uses Yices's `--timeout` option.

4.10 Z3 options

No Z3 options are specifically supported by Victor. Z3 options can be specified by giving a custom prover command with the `prover-command` option.

5 Translation

The description of the translation process here is rather brief and not self-contained. The process is best understood by first having a read of the draft paper *Proving SPARK Verification Conditions with SMT solvers*, available from the author's website.

Unless otherwise stated, translation steps are carried out in order they are described in below.

5.1 Standard Form translation

Most translation steps in Victor are carried out on units in a standard form. In this standard form all functions and relations have a unique type, there is no overloading.

The first translation step is to put units into this standard form.

- Some constants with names of form *c*`_base_first` or *c*`_base_last` are used but not declared. Victor adds declarations for such constants when they appear to be missing, when e.g. the constant *c*`_first` is declared (*c* not with suffix `_base`) and the constant *c*`_base_first` is not declared.
- The FDL files output by the Examiner are missing declarations of the *E*`__pos` and *E*`__val` functions used by each enumeration type *E*, including the implicitly declared `character` type. These declarations are added in.
- FDL variables are considered as semantically the same as FDL constants. Each declaration of an FDL variable x, is changed to a constant declaration, and new declarations are added for names x`~` and x`%`. FDL units use the names x`~` and x`%` to refer to the value of x at procedure and loop starts respectively.
- Occurrences of the FDL operator `sqr(x)` are replaced by `x ** 2`.
- Distinct operators are introduced for the standard arithmetic operations $+$, $\times$, $-$(unary), $-$(binary) over the integers and reals, and an explicit coercion operator is introduced for converting integers to reals.
- Distinct relations are introduced for the inequality relations over integers, reals, and enumeration types.

- Distinct versions of the FDL operator `abs(x)` are introduced for the real and integer types. Defining axioms are added to the set of rules for each unit.
- A defining axiom is added for the FDL predicate `odd(x)`.
- Some characterising axioms are added for the FDL operator `bit__or(x)`. No axioms are added yet for other bit-wise arithmetic operators.
- The FDL language overloads the functions `succ` and `pred` and inequality relations such as $<$ and $\leq$. Distinct versions are introduced for the FDL `integer` type and each enumeration type and declarations are added for each of these versions.
- The Examiner outputs rules with implicitly quantified variables. Victor infers the types of these variables and makes the quantifications explicit. The explicit quantification is needed by all the provers to which Victor interfaces.

5.2 Type checking

Victor type checks units after translation into standard form and after all translation steps have been applied.

5.3 Enumerated type abstraction

`-abstract-enums`
: Replace enumerated types with abstract types, introduce all enumeration constants as normal constants, and keep all enumerated type axioms.

`-elim-enums`
: Replace each enumeration type E with an integer subrange type $\{0 \ldots k-1\}$ where k is the number of enumeration constants in E. Declare each enumeration constant as a normal constant, and add an axiom giving its integer value. Delete all existing enumerated type axioms and add in new axioms characterising enumerated-type-related functions such as E`__val` and E`__pos`.

5.4 Early array and record abstraction

`-abstract-arrays-records-early`
: Enable abstraction at this point

See Section 5.7 below for rest of options

5.5 Separation of formulas and terms

In FDL formulas are just terms of type Boolean. Many provers require the traditional first-order logic distinction between formulas and terms. The options here control the introduction of this distinction.

Victor calls the term-level Booleans *bits*.

`-bit-type`
: Enable separation.

`-bit-type-bool-eq-to-iff`
: Initially convert any equalities at Boolean type to ‘if and only if’s.

`-bit-type-with-ite`
: Whenever possible, introduce instances of the ‘if-then-else’ operator rather than term-level versions of propositional logic operators and atomic relations.

`-bit-type-prefer-bit-vals`
: A heuristic for controlling whether atomic relations are translated to term-level (bit-valued) functions or first-order-logic formula-valued relations. With this heuristic, bit-valued functions are preferred.

`-bit-type-prefer-props`
: Another heuristic for controlling whether atomic relations are translated to term-level (bit-valued) functions or first-order-logic formula-valued relations. With this heuristic, formula (propositional) relations are preferred.

  If neither this option or `-bit-type-prefer-bit-vals` is selected, the default behaviour is to use a bit-valued function just when there is one or more occurrences at the term level.

`-trace-prop-to-bit-insertion`
: Report in log file when a proposition-to-bit coercion (encoded using the ‘if then else’ operator) is added.

`-trace-intro-bit-ops-and-rels`
: Report in log file when term-level function is introduced for a function (either user-defined or built-in) that initially had Boolean value type.

NB: the SPARK FDL language has ‘bit’ operators `bit__or`, `bit__and` and `bit__xor`. These FDL operators take integers as arguments and return integers as results. Their result values correspond to the correct unsigned binary result for the respective operations on unsigned binary versions of the arguments. Axioms on these operators capture the arithmetic properties of Boolean operations on finite-length binary words. If the Victor option `-abstract-bit-ops` is used, Victor introduces operators `bit___or`, `bit___and` and `bit___xor`. These operators work on the term-level Booleans introduced by Victor and are distinct from the SPARK FDL bit operators.

5.6 Type refinement

`-refine-types`
: Master control

`-refine-bit-eq-equiv`
: Add in definition for bit-valued non-trivial equivalence relations. Needed when `-bit-type-with-ite` option not previously selected.

`-refine-int-subrange-type`

`-refine-bit-type-as-int-subtype`

`-refine-bit-type-as-int-quotient`

`-refine-array-types-with-quotient`

`-refine-array-types-with-weak-extension-constraint`
: Constrain values of element and extended indices using possibly non-trivial equivalence relation on element type. Default is to use equality to constrain these values.

  Only applies if option `-refine-array-types-with-quotient` is not selected.

`-refine-uninterpreted-types`
: Refine every uninterpreted type to be predicate subtype of a new uninterpreted type. Use this to ensure that exists model in which every uninterpreted type can be interpreted by some infinite set.

`-no-subtyping-axioms`
: Suppress generation of axioms for sub-typing properties of functions and constants.

`-no-functionality-axioms`
: Suppress generation of axioms for functionality properties of functions and relations.

`-strong-subtyping-axioms`
: Use subtyping axioms without constraints on values of arguments.

`-trace-refine-types-quant-relativisation`
: Report in log file whenever a quantifier is relativised.

`-trace-refine-types-eq-refinement`
: Report in log file whenever an equality relation is refined to a non-trivial equivalence relation.

`-trace-refine-types-bit-eq-refinement`
: Report in log file whenever an term-level equality relation is refined to a non-trivial term-level equivalence relation.

5.7 Late array and record abstraction

`-abstract-arrays-records-late`
: Enable abstraction at this point in translation.

`-elim-array-constructors`
: Eliminate all occurrences of array constructors

`-elim-record-constructors`
: Eliminate all occurrences of record constructors

`-abstract-record-updates`
: Introduce axiomatic characterisations for record update operators in terms of record constructors and record field selectors.

`-add-array-select-update-axioms`
: Assumes that array constructors have first been eliminated.

`-add-array-extensionality-axioms`

`-add-record-select-constructor-axioms`
: Assumes that record update operators have first been eliminated.

`-add-record-constructor-extensionality-axioms`
: Add extensionality axioms involving record constructors and field select operators.

`-add-record-select-update-axioms`
: Assumes that record constructors have first been eliminated.

`-add-record-eq-elements-extensionality-axioms`
: Add extensionality axioms stating that records are equal just when all fields are equal.

`-use-array-eq-aliases`
: Introduce aliases for equalities at array types in order to help with matching extensionality axioms.

`-use-record-eq-aliases`
: Introduce aliases for equalities at record types in order to help with matching extensionality axioms.

`-abstract-array-select-updates`
: Change primitive array element select and update operators into uninterpreted functions.

`-abstract-array-types`
: Replace array types with uninterpreted types.

`-abstract-record-selects-constructors`
: Change primitive record field selectors and constructors into uninterpreted functions.

`-abstract-record-selects-updates`
: Change primitive record field selectors and field update operators into uninterpreted functions.

`-abstract-record-types`
: Replace record types with uninterpreted types.

5.8 Bit abstraction

`-abstract-bit-ops`
: Replace primitive bit-type operators with uninterpreted functions and add characterising axioms

`-abstract-bit-valued-eqs`
: Replace primitive bit-valued equality operators with uninterpreted functions and add characterising axioms

`-abstract-bit-valued-int-le`
: Replace primitive bit-valued integer inequality operators with uninterpreted functions and add characterising axioms

`-elim-bit-type-and-consts`
: Replace primitive bit type with either integer type or $\{0\,..\,1\}$ subrange type, depending on whether type has been refined earlier or not. Replace primitive bit-type constants for true and false with 0 and 1.

5.9 Arithmetic simplification

`-elim-consts`
: Eliminate integer constants. Rewrite all formulae using hypotheses of form $x = k$ or $x = -k$ where x is an FDL constant or variable, and k is a natural number literal. This eliminates the apparent syntactic non-linearity of some hypotheses and conclusions. It is particularly useful for Yices which rejects formulae that appear non-linear.

`-ground-eval-exp`
: Evaluate occurrences of exponent function with natural number arguments.

`-ground-eval`
: Evaluate ground integer arithmetic expressions involving $+$, $-$ (unary and binary), $\times$, integer division, integer modulus, and the exponent function.

`-expand-exp-const`
: Expand natural-number powers of integer and real expressions into products, with special-case treatment for exponents 0 and 1.

`-arith-eval`

Apply the rewrite rules

$$\begin{aligned}
k \times (k' \times e) &= kk' \times e \\
(k \times e) \times k' &= kk' \times e \\
e \times k &= k \times e \\
(k \times e) \times (k' \times e') &= kk' \times (e \times e') \\
e \times (k \times e') &= k \times (e \times e') \\
(k \times e) \times e' &= k \times (e \times e') \\
(k \times e) \div k' &= (k \div k') \times e \quad \text{if } k' \text{ divides } k.
\end{aligned}$$

The main aim of these rules is to eliminate instances of the $\div$ operator.

`-sym-consts`

Replace each distinct natural number literal greater than threshold t with a new constant and assert axioms concerning how these new constants are ordered: if the new constants in increasing order are $c_1 \ldots c_n$, the axioms are $t < c_1, c_1 < c_2, \ldots, c_{n-1} < c_n$.

This option is used to try to reduce the frequency of machine arithmetic overflow with Simplify. Other users of Simplify try thresholds of 100,000, though we've observed overflows with thresholds as low as 1000.

`-sym-prefix=`*prefix*

Set prefix for new symbolic number constants. Default prefix is `k___`.

5.10 Arithmetic abstraction

The different interfaces and provers vary in the classes of arithmetic operations they can handle. These options allow one to abstract to uninterpreted functions, possibly adding some characterising axioms, when operations cannot be handled.

`-abstract-nonlin-times`

Abstract each integer and real multiplication unless at least one of the arguments is a fixed integer or real constant.

An *integer constant* is a natural number n or the expression $-n$.

A *real constant* is built from an integer constant using the `to-real` coercion, unary minus on the reals, and, optionally real division. Real division is allowed just when the option `-abstract-real-div` is not chosen.

The Yices API usually rejects individual hypothesis or conclusion formulas if they have non-linear multiplications. However, it does accept non-linear multiplications in quantified formulas, and will use linear instantiations of these formulas. Unfortunately, it currently aborts on finding a non-linear instantiation rather than simply rejecting the instantiation.

The SMT-LIB sub-logics AUFLIA and AUFLIRA both require all multiplications to be linear.

`-abstract-exp`

Replace occurrences of integer and real exponent operators by new uninterpreted functions. Currently no defining axioms are supplied, though it would be easy to do so.

This abstraction only happens after possibly evaluating ground and constant exponent instances.

Only the Cvc3-via-API prover alternative can handle these operators directly.

`-abstract-divmod`

Replace occurrences of integer division and modulus operators by new uninterpreted functions.

`-abstract-real-div`

Abstract occurrences of real division to a new uninterpreted function. No characterising axioms are currently provided.

Yices-API, Cvc3-API and Z3-SMT-LIB all allow input with the real division operator, though it is not known what kinds of occurrences are accepted in each case.

The official SMT-LIB logics involving reals do not allow real division. The assumption is that pre-processing has eliminated all occurrences of real division. Victor doesn't yet carry out such pre-processing.

`-abstract-reals`

Abstract occurrences of real arithmetic operations ($+$, unary $-$, binary $-$, $*$, $/$), integer to real coercions, and real inequalities to new uninterpreted functions.

Currently this is needed by the SMT-LIB and Simplify translations. The SMT-LIB driver does not attempt to make use of the limited support in some of the SMT-LIB sub-logics for reals.

This option is not necessary when Cvc3 and Yices are invoked via their APIs, as both APIs support real arithmetic.

5.11 Final translation steps

`-elim-type-aliases`

Normalise all occurrences of type identifiers in type, constant, function and relation declarations and in all formulas. Normalisation eliminates all occurrences of type ids T that have a definition $T \doteq T'$ where T' is either a primitive atomic type (Boolean, integer, integer subrange, real or bit type) or is itself a type id.

This is needed for the SMT-LIB and Simplify translations.

`-switch-types-to-int`

Replace all occurrences of type identifiers in constant, function and relation declarations and in all formulas with the integer type. Checks that every defined type is either an alias for another defined type or an alias for the integer type. This translation step assumes that a countably infinite model exists for every uninterpreted type.

This option is is needed for the Simplify translation.

`-lift-quants`
: Apply the rewrite rule

$$P \Rightarrow \forall x : T.\ Q \quad \Leftrightarrow \quad \forall x : T.\ P \Rightarrow Q$$

(x not free in P) to all formulae. The quantifier instantiation heuristics in both Z3 and Simplify work better when universal quantifiers in hypotheses are all outermost.

`-strip-quantifier-patterns`

Some of the universally-quantified axioms introduced by translation have trigger patterns giving hints on how instantiations can be guessed. This option strips out these patterns.

6 CSV utilities

These utilities are very useful for analysing and comparing results of Victor runs.

6.1 Filter CSV records

Usage:

`csvfilt` [`-v`] *n str* [*file*]

Filter VCT records, returning on standard output just those with *str* a substring of field *n* (1-based). If `-v` provided, then return those records without *str* a substring of field *n*. Records are drawn from file *file* if it is supplied. If not, they are taken from standard input.

6.2 Merge two CSV files

Usage:

`csvmerge` *file1 m1 . . . mj file2 n1 . . . nk*

The files *file1* and *file2* must have the same number of records. This command merges corresponding records from the two files and outputs them on standard output. The merged records are composed from fields *m1 ... mj* in the records in *file1* and fields *n1 ... nk* in the records in *file2*. If $j = 0$, all fields of *file1* records are used. If $k = 0$, all fields of *file2* records are used.

6.3 Project out fields of CSV records

Usage:

`csvproj` *n1 ... nk* [*file*]

Build new records from fields *n1 ... nk* of the input records and output to standard output. Input records are drawn from file *file* if it is supplied. If not, they are taken from standard input.

Usage:

`csvisect` *file1 file2*

Print on standard output those records that occur in both *file1* and *file2*. Comparison of records currently just uses string equality, so it is sensitive to whitespace between record fields.

7 Future developments

Anticipated by end of 2010 are

- Support for V2 of the SMT-LIB standard.

  One major benefit of this will be better support for VCs involving mixed real and integer arithmetic. Currently Victor does translate real arithmetic to V1.2 of the SMT-LIB format. However, V1.2 does not support well goals in which integers and reals are mixed. (For example, it does not define a function injecting the integers into the reals.)

- Support for outputing VCs for proof using the Isabelle/HOL theorem prover.

  The current release includes some preliminary code for this. Improved code has been developed and is waiting to be merged in.

- A fully-working Windows port.

  The current release has some code for this, but it is not yet fully tested.