Victor: a SPARK VC Translator and Prover Driver

User Manual for release 0.9.1
and subsequent experimental modifications

Paul Jackson
pbj@inf.ed.ac.uk

11 Feb 2013

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, SMT-LIB v1.2 or SMT-LIB v2 format input files.

1.1 Simplify language

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

1.2 SMT-LIB languages

The SMT-LIB initiative (http://www.smtlib.org) defines standard languages for formatting input to SMT solvers and collects benchmarks in this format. Victor supports the current version 2.0 and the older version 1.2.

The initiative defines particular *logics*, each characterised by some combination of logical theories and possible restrictions on shapes of terms and formulas. These logics are used to classify benchmarks, and different SMT solvers support different logics. Some solvers support a superset of what is defined by any of these logics.

At a minimum, Victor requires solvers to handle uninterpreted functions and constants, linear integer arithmetic and logical quantifiers.

The following logics in both v1.2 and v2.0 provide at least this support:

- 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.

Victor can also take advantage of support for real arithmetic and for sort symbols. Such provers include Alt-Ergo, CVC3, CVC4, Yices and Z3.

One major benefit of v2.0 is better support for queries involving mixed real and integer arithmetic. 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.)

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 v0.95 release (11th Jan 2013) using Alt-Ergo's SMT-LIB 1.2 and 2.0 file-level interfaces. Alt-Ergo's 1.2 interface works fine, but issues have been noted with the 2.0 interface in both the v0.94 and v0.95 releases.

1. `define-type` is not supported.
2. `to_real` function is not recognised.
3. Instances of quantified Bool-typed variables are not recognised when they occur in formula positions. For example, it rejects

   ```
   (assert (forall ((x Bool)) x))
   ```

   though it accepts

   ```
   (assert (forall ((x Bool)) (= x x)))
   ```

4. Boolean operators ('and', 'or', etc.) are not recognised when they occur in term positions. For example, it rejects

   ```
   (declare-fun F (Bool) Bool)
   (assert (F (and true true)))
   ```

Victor has full work-arounds for 1, 3 and 4 and a partial work-around for 2. For 2, it replaces instances of `to_real` with an uninterpreted function. Axiomatisation of the properties of `to_real` has not yet been explored.

Alt-Ergo's output is also not fully compliant with both the 1.2 and 2.0 standards: it will output the non-standard `unknown (sat)` rather than simply `sat`, and output multiple `unsat` responses to a single satisfiability check request. Victor has work-arounds so it tolerates these non-standard issues.

The v0.95 release is preferred over the v0.93 and v0.94 releases as both v0.93 and v0.94 sometimes has been observed to have fatal errors, whereas no such errors have been observed yet with v0.95.

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 last release, V2.4.1, dating from September 2011. CVC3 is significantly slower than Yices or Z3 (maybe 5-10×), especially when VCs are unprovable. It has some basic support for non-linear arithmetic.

The API interface of this version is essentially unusable by Victor, as, on some goals, the CVC3 code ignores the request for it to return after a set timeout period. When this happens, Victor itself is just stuck and cannot progress on to other goals.

1.5 CVC4

CVC4 is a successor to CVC3, developed by the same team. It is available from http://cvc4.cs.nyu.edu.

Victor has been tested with the first release, V1.0, made on 3rd December 2012, using the the SMT-LIB 1.2 and 2.0 file-level interfaces. Currently there is no API interface to CVC4.

While CVC4 does not support the `to_real` operator in mixed integer real SMT-LIB 2 logics, it does seem to allow implicit integer to real conversion which achieves much the same end.

1.6 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`. These are 32-bit executables. No success has been found in locating any 64-bit executables.

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.

1.7 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, V1.0.37 (Jaunary 2013). This version, bug fixes apart, dates from summer 2007 and essentially is the version that led 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 1.2 format files. In this case all arithmetic must be linear – non-linear arithmetic operators have to be abstracted to uninterpreted functions.

A successor solver Yices 2 is available. The latest version is 2.1.0, released August 2013. This version is unsuitable for use as it does not support quantifiers.

1.8 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 4.3.1, made on 12th Nov 2012. 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 SMT-LIB 1.2, SMT-LIB 2.0 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 currently compiled and run on a Scientific Linux 6 64-bit platform. Scientific Linux 6 is based on Red Hat Enterprise Linux 6. It makes use of the following tools:

- `make` V3.81
- `gcc/g++` V4.4.5
- `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, VUS and VSM 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. Build an auxiliary program for timing-out runs of provers. Enter `make watchdog`. This creates the executable `run/watchdogrun`.

7. 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. For brevity, we sometimes collectively refer to SPARK VC goals and slices of SPARK VC goals as *Victor goals* or even just *goals*.

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. For each program unit
   (a) Read in the VCs described in the `.fdl`, `.rls`, `.vcg` file triples output by the SPARK Examiner for the unit.[1]
   (b) Invoke a prover on each goal or goal slice.
   (c) Output `.vct`, `.vus`, `.vsm` and `.vlg` report 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

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

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 Goal report (VCT) output file

The goal report or VCT file has suffix `.vct`.

The file is in comma-separated-value (CSV) format: linebreaks divide the file into records, and each record is divided by commas into fields. Field values in general might contain commas and linebreaks, in which case the field value is enclosed in `""`s.

Each record gives information on one Victor goal. The record fields are:

1. Unit number. The number of the unit the goal is from. Victor numbers the units it reads sequentially, starting from 1.
2. Path to unit. This describes the containing packages.
3. Unit name, without a prefix for the containing packages.
4. Unit kind. One of `procedure`, `function` or `task_type`.
5. Goal origins information – source of path in subprogram for VC .
6. Goal origins information – destination of path for VC, or VC kind if not path related.
7. VC goal number.
8. Conclusion (goal slice) number. Empty if the Victor goal is generated form the whole SPARK goal.
9. Status (one of `trivial`, `true`, `unproven` or `error`).
10. Proof time (in sec).
11. Brief remarks about goal and solver interactions
12. Operator kinds occurring in hypotheses
13. Operator kinds occurring in conclusion

A file `vct-file-header.txt` provides a 1 line comma-separated list of field headings for these goal report files. By default, the VCT files are generated without this heading.

3.3.3 Unit summary (VUS) output file

The unit summary or VUS file has suffix `.vus` and is in CSV format. Each record gives a summary of the Victor run on one unit. The record fields are:

1. Unit number.
2. Unit name, including prefix showing containing packages
3. Number of `ERROR` messages written to VLG log file.
4. Number of `WARNING` messages written to VLG log file.
5. Total number of goals.
6. Number of trivial goals.

7. Number of goals found true by the prover.
8. Number of goals unproven by the prover.
9. Number of unproven goals where the prover timed out.
10. Number of unproven goals where the prover asserted the goal was definitely false.
11. Number of goals where some error occurred in handling the goal
12. Number of user rules read in from the directory-level and unit-level user rule (`.rlu`) files.
13. Number of directory-level user rules excluded because of unbound function, constant or type ids, or because they do no not type check.
14. Number of unit-level user rules excluded.
15. Number of system rules excluded, (*system rules* being rules generated by the Examiner or provided by Victor).
16. Size of combined parse tree for unit input files.
17. Size of abstract syntax tree after all prover-independent translation steps in Victor.
18. Number of abstract syntax tree nodes allocated for unit
19. Total time to handle unit
20. Total time in prover when proof succeeded.
21. Average time in single prover run when proof succeeded.
22. Maximum time in single prover run when proof succeeded.
23. Total time in prover when proof failed.
24. Whether unit is inconsistent, when all user rules and all hypotheses and conclusions of specific goals are ignored. Value is 1 if inconsistent, otherwise 0.
25. Whether unit is inconsistent, when all user rules considered, but all hypotheses and conclusions of specific goals are ignored. Value is 1 if inconsistent, otherwise 0.
26. Number of user rules found to be inconsistent with system-supplied rules and built-in theories of the prover.
27. Number of user rules found to be derivable from system-supplied rules and built-in theories of the prover.
28. Number of user rules found to be derivable from other user rules, along with system-supplied rules and built-in theories of the prover.
29. Remarks on unit generated by Victor

A file `vus-file-header.txt` provides a 1 line comma-separated list of field headings for these unit summary files. By default, the VUS files are generated with this heading to simplify reading the files in a spreadsheet program.

3.3.4 Session summary (VSM) output file

The session summary or VSM file has suffix `.vsm` and is in CSV format. It contains one record summarising a Victor run over one or more units. The record fields are:

1. Report file name with any of the suffices.
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 goals with *true* status
6. Number of goals with *unproven* status
7. Number of unproven goals that involved a timeout.
8. Number of unproven goals that were shown to be definitely false.
9. Number of goals with *error* status
10. Percent of goals with *true* status
11. Percent of goals with *unproven* status
12. Percent of goals that were unproven and involved a timeout.
13. Percent of goals that were shown false.
14. Percent of goals with *error* status
15. Total time to handle all units of session
16. Total time in prover when proof succeeded.
17. Average time in single prover run when proof succeeded.
18. Maximum time in single prover run when proof succeeded.
19. Total time in prover when proof failed.
20. Number of inconsistent units, when all user rules and all hypotheses and conclusions of specific goals are ignored. Value is 1 if inconsistent, otherwise 0.

21. Number of inconsistent units, when all user rules considered, but all hypotheses and conclusions of specific goals are ignored. Value is 1 if inconsistent, otherwise 0.
22. Number of user rules found to be inconsistent with system-supplied rules and built-in theories of the prover.
23. Number of user rules found to be derivable from system-supplied rules and built-in theories of the prover.
24. Number of user rules found to be derivable from other user rules, along with system-supplied rules and built-in theories of the prover.

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 the header file and then viewed in any spreadsheet program.

3.3.5 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. user rule analysis reports,
4. 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.

`-read-all-decl-files-in-dir`
: For each program unit with full name *dirs/dir/unitname*, read in all FDL declaration files in directory *dirs/dir/* rather than just the FDL declaration file *dirs/dir/unitname.fdl*. This option is useful in conjunction with `-read-directory-rlu-files` and `-read-unit-rlu-files` when the user-defined rules files for a unit (see options `-read-directory-rlu-files` and `-read-unit-rlu-files`) refer to identifiers that are not declared in *dirs/dir/unitname.fdl*. An alternative option is `-expect-dir-user-rules-with-undeclared-ids`.

`-read-directory-rlu-files`
: For each program unit with full name *dirs/dir/unitname*, read in the user-defined rules file *dirs/dir/dir.rlu*. The SPARK Simplifier reads in such rules files by default.

`-read-unit-rlu-files`
: For each program unit with full name *dirs/dir/unitname*, read in the user-defined rules file *dirs/dir/unitname.rlu*. The SPARK Simplifier reads in such rules files by default.

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

`-expect-dir-user-rules-with-undeclared-ids`
: Directory-level user-defined rules files can sometimes include rules that involve type, function and constant identifiers that are not declared in the FDL file for a unit being processed. Normally, when Victor encounters a rule that has some undeclared identifiers, it deletes that rule and outputs a warning message to the log file. With this option, an info message rather than a warning message is output for directory-level user rules involving undeclared ids. An alternative option is `read-all-decl-files-in-dir`.

`-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.

`-skip-rule=`*r*
: Do not pass rules with name matching pattern *r* to the selected prover. The pattern *r* can contain wildcards

  - '`*`' matching 0 or more characters in the name, and
  - '`?`' matching exactly one character in the name.

  The simple matching algorithm requires that any instance of '`*`' in *r* be at the end of *r* or be followed by some character other than '`*`' or '`?`'. If '`*`' is followed by some character *c*, then the sequence of characters matched by '`*`' will not contain *c*.

`-skip-check`
: Skip issuing command to prover to run a check of the asserted goal. This has been found helpful when debugging problems with the selected prover.

`-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.

`-skip-units-with-no-rlu-files`
: Run only on units with RLU files. This is particularly useful when doing user rule analysis. See Section 4.4 below.

`-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 Normalisation options

Normalisation is the process of putting an FDL unit into a standard form where all ambiguities have been resolved. Specifically, normalisation

- resolves the types of overloaded operators:
  - `+`, `-` (binary), `-` (unary), `*` on integer and real types,
  - `=` and `<>` on any type,
  - `<` , `<=`, `>` and `>=` in integer, real and enumeration types,
  - `succ` and `pred` on integer and enumeration types,
  - array element update and element select functions,
  - record field update and field select functions,
- adds universal quantifiers to rules to bind and type any free variables in the rules.

After normalisation but before any translation steps are applied, Victor does a full type check of the unit to ensure the integrity of the unit. If normalisation has not been completely successful, type check error messages identify the issues.

Overload resolution is driven by the inferred types of each overloaded operator's arguments. So long as the argument types can indeed be inferred, overload resolution is straightforward. However, when an argument is a free variable in a rule, sometimes Victor has difficulty inferring which operator is intended, in which case overload resolution fails.

Victor infers the type of a free variable by examining the typing of the operator applied to each instance of the variable. This works fine if the operator is not overloaded.

In the first phase of ambiguity resolution, Victor repeatedly interleaves

- attempts at operator overload resolution, and
- inference of constraints on types of free variables

until no further progress can be made. This phase is always sound. In this phase Victor will never speculatively resolve overloading or speculatively choose some type for a free variable.

As mentioned above, if the ambiguity resolution has not been completed, the subsequent type check will identify the issues. To date, all the observed issues have had at their root the failure to infer types of free variables in user rules provided in RLU files. Invariably the problem is with free variables which only appear as arguments to overloaded operators. To fix the issues, add extra explicit typing assumptions of form `goal(checktype(`*`var, type`*`))` to the rules identified in the error messages. Usually the Simplifier tool handles these typing assumptions properly. However the recommended practice is that it always be re-run after these assumptions have been added in order to double check them.

This addition of explicit typing assumptions can require a fair amount of work, especially if there are 100s of user rules and the FDL involves reasoning with operators over the reals. For example, Victor will observe that every free variable that could be typed `real` could also be typed `integer`, and will therefore refuse to assign the variable a type in this first phase.

By default ambiguity resolution stops at the first phase. However, Victor has three options which allow the user to tell Victor to go onto a second phase of ambiguity resolution where extra assumptions are made. The options are:

`-assume-var-in-real-pos-is-real`
: If a free variable occurs as an argument to an operator in a position where a real is expected, assume the variable has real type.

`-assume-int-or-real-var-is-real`
: If a free variable is constrained to have either integer or real type, assume the variable has real type. This option subsumes the previous option.

`-assume-int-or-real-var-is-int`
: If a free variable is constrained to have either integer or real type, assume the variable has integer type.

At most one of these 3 options should be selected. The first two options are potentially unsound. To avoid unsoundness when either of these options is used, any free variable that ought to be integer must be explicitly typed by a `checktype` precondition. The third option is always sound, but could result in rules that are overly restrictive and not usable as expected by the Simplifier. One situation where the 3rd option might be appropriate is when SPARK programs being analysed involve no floating-point computations, and hence when the generated VCs involve no terms of real type.

When any of these options is selected, Victor by default will insert a warning message into the VLG log file for each assumption it makes, so each application of the assumption can be checked. If these warning messages are not needed, use the following option.

`-suppress-warnings-of-var-type-assumptions`

Suppress warnings of each application of an assumption about how a free variable should be typed.

4.3 Translation options

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

4.4 User rule analysis

These options analyse the rules that users add to the directory-level and unit-level .rlu user rule files. Users add rules to overcome incompletenesses in the Simplifier prover and to provide extra information that cannot easily be routed through the Examiner. These options help identify those rules required by the Simplifier, but not some SMT solver. They also help to pick out rules that inadvertently are inconsistent.

4.4.1 Finding redundant rules

`-find-redundant-rules`

Consider each goal that is provable with all the user rules, and determine a minimal set of user rules needed to prove the goal.

- Report on which unit-level user rules are not in any of the minimal sets of the goals for a given unit. These unit-level user rules are redundant.
- Across all the goals in the units in a given directory, report on which directory-level user rules are are not in any of the minimal sets for each goal in each unit.

The minimal set for a goal is found in a greedy fashion, starting with the full set of user rules, and then seeing if each rule in turn can be knocked out without affecting the provability of the goal. The minimal set is similar to a proof summary / unsat core see the documentation on the `-smtlib2-unsat-cores` option in Section 4.6). However, an unsat core might well include a user rule in a core because the SMT solver happened to first discover a proof using the rule, even if there are proofs without the rule. Here, when a user rule is considered, it is retained only if it is seen to be definitely needed.

`-report-user-rule-status`

For each goal, report the .vlg log file on the status of each user rule, whether it was

1. excluded from consideration because of unbound function, constant or type identifiers,
2. considered and found to be outside the minimal set for that goal,
3. considered and found to be inside the minimal set.

4.4.2 Auditing rules

In a *rule audit*, for each unit Victor completely discards the goals read in from .vcg or .siv files and instead generates a set of goals that check various properties of the user rules. The newly generated goals are divided into 5 kinds, as shown in Table 1.

Kind	Goal shape	Description	.vct tag	
A	$R \vdash \bot$	Are system rules inconsistent?	`no urules`	
B	$R, U \vdash \bot$	Are user rules inconsistent?	`all urules`	
C	$R, u_i \vdash \bot$	Is user rule u_i inconsistent?	`urule as H`	u_i-name
D	$R, \vdash u_i$	Is user rule u_i derivable with no use of other user rules?	`urule as C`	u_i-name
E	$R, U \setminus \{u_i\} \vdash u_i$	Is user rule u_i derivable from other user rules?	`urule from rest`	u_i-name

Table 1: Kinds of Rule Audit Goals

R is the set of system rules, either generated by the Examiner and read in from an .rls file, read in by Victor from one of its standard rules files (e.g. `prelude.rul`), or generated by Victor as part of its translation, and $U = \{u_1, \ldots, u_n\}$ is the set of user rules read in from directory-level and unit-level user rule files, excluding those directory-level rules that are not well formed because they have unbound constant, function or type identifiers. For each unit, 1 goal of each of kinds A and B is generated, and n goals of each of kinds C,D and E are generated. The two columns labelled *.vct tag* show what is output to the *goal origins* fields in the goal-based .vct report file.

Victor writes the number of goals of each kind that found to be true to the .vus unit summary file, the .vsm session summary file and in an extra audit report at the end of the .vlg log file.

The options are as follows.

`-do-rule-audit`
: Generate audit goals and collect results in report files.

`-rule-audit-a`
: Include goals of kind A in audit.

`-rule-audit-b`
: Include goals of kind B in audit.

`-rule-audit-c`
: Include goals of kind C in audit.

`-rule-audit-d`
: Include goals of kind D in audit.

`-rule-audit-e`
: Include goals of kind E in audit.

`-rule-audit-rule=`*rname*

Generate audit goals just for rule with name *rname* rather than for all of $u_1, \ldots, u_n$. This option makes sense when attention is focussed on one unit or, if a directory-level rule is named, all units in a single directory.

4.5 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 1.2 format files and stand-alone prover executable. Acceptable with `cvc3`, `yices` or `z3` value for `prover`, and with `prover-command` option.
- `smtlib2`: Use SMT-LIB 2.0 format files and stand-alone prover executable. Acceptable with `cvc3` 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, VUS, VSM and VLG output files. This is the default option.

4.6 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.

`-delete-working-files`
: Delete the files used for prover input and output after each prover invocation.

`-add-formula-descriptions`
: Add comments in the prover input file describing the rule, hypothesis or conclusion that each asserted formula comes from.

`-smtlib2-unsat-cores`
: Ask the prover to report an unsatisfiability core for each proven goal. An *unsatisfiability core* or *unsat core* names which of the rules, hypotheses and the conclusion of a goal were used in the proof of the goal. An unsat core can be viewed as a proof summary. Unsat cores are reported in the remarks field of records written to the VCT file.

  Unsat cores are currently only supported if the SMT-LIB2 file-level prover interface is used.

  Generation of unsat cores can slow down provers. For example, Z3's run-time on provable goals has been observed more than double when unsat core generation has been requested.

`-smtlib2-add-to_real-decl`
: Add a declaration for the SMT-LIB 2 `to_real` operator. This is needed for Alt-Ergo.

`-smtlib2-implicit-to_real`
: Remove all occurrences of the SMT-LIB 2 `to_real` operator in SMT-LIB 2 format files created by Victor. The SMT-LIB 2 standard allows these operators to be implicit. This option is needed for supporting CVC4 1.0.

`-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.

`-watchdog-timeout=`*time*

If using the SMT-LIB2 file-level interface mode, use the provided *watchdogrun* C program to time out prover invocations after *time* seconds of inactivity on the prover's output. The `time` value can be a natural number or a fixed-point number (e.g. 0.1). The default is not to time out prover invocations.

The timeout measurement from the last output activity (or prover start time) is useful when a single call of the prover is used to answer multiple queries. When timeouts are measured this way, they bound the run-time of the prover on the current query it is attempting, rather than also including in the allowed run period the times spent by the prover answering previous queries in a query set.

`-smtlib2-soft-timeout=`*time*

If using the SMT-LIB2 file-level interface mode and Z3 v3.1 or newer, timeout *check-sat* prover invocations after *time* milliseconds. The `time` value should be a natural number.

`-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 a natural number 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.

`-use-alt-solver-driver`

This option enables an alternative driver. The new major features supported by this driver are *incrementality* (see next option) and *user rule analysis* (see Section 4.4).

`-exploit-solver-incrementality`

This option gives significant reductions in prover run-time.

Prior to summer 2011, the main solver driver always handed goals to the solver one at a time. Each solver invocation had a complete separate copy of all the declarations and rules, even though these are constant across the goals of a unit.

With this option, when driving a unit, declarations and rules are passed to the solver just once, rather than repeatedly for each goal. Use is made of solver support for pushing and popping contexts.

This option has been tested with the SMT-LIB 2 file-level interface. It ought to work with both the Yices and CVC3 API interfaces, though this has not yet been fully enabled.

When using the file-level SMT-LIB 2 interface, the driver initially creates a single SMT-LIB 2 format file containing queries for all the goals in a unit. If the solver is killed because it timed out on some goal, a new solver input file is created just for the remaining goals. On each further timeout, a further new solver input file is created.

This option is best used in conjunction with `-watchdog-timeout` or `smtlib2-soft-timeout` to ensure timeouts are applied to each individual query. The `smtlib2-soft-timeout` option gives better performance, as the solver is then not killed on timeouts, and after a timeout can continue on further goals from the same input file: there is no need for rerunning the solver on a new file containing the remaining goals. However, for some reason, when run with the same time limits, the coverage with the soft timeout is lower. This is currently being investigated.

4.7 Output options

4.7.1 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.

4.7.2 General report file options

`-report=`*report-file*

Use *report-file* as body of filenames for VCT, VUS, 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.

4.7.3 VCT file options

`-count-trivial-goals`

Write an entry to the VCT file for each SPARK 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 the status `trivial` recorded in the status field.

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 VCT file records.

`-ckinds`
Report list of concl kinds in VCT file records.

`-gstime`
Report time taken by prover to process a goal in time field of VCT file records.

`-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 goal origins fields of VCT file records. Default is not to include this information.

`-csv-reports-include-unit-kind`
Include information on unit kind in unit kind field of VCT file records. For example, whether the unit is a function or a procedure. Default is not to include this information.

4.7.4 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*.

4.8 Debugging options

`-scantrace`
Write lexer debugging information to standard output

`-parsetrace`
Write parser debugging information to standard output

4.9 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.

4.10 Simplify options

No options are currently available.

4.11 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.

4.12 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 Verifi-*

cation 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. These axioms are introduced by the Examiner to characterise enumerated-type-related functions such as E`__val` and E`__pos` and can serve as a partial axiomatisation of the introduced abstract types.

`-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`.

`-axiomatise-enums`
: Replace each enumeration type E with an uninterpreted type, and add axioms characterising the the uninterpreted type as isomorphic to the integer subrange $\{0 \ldots k-1\}$ where k is the number of enumeration constants in E. The added axioms replace the enumerated-type-related axioms introduced by the Examiner and provide a full axiomatisation of the enumerated types.

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-real-le`
: Replace primitive bit-valued inequality operators on integers and reals 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 and real constants. Rewrite all formulae using hypotheses of form $x = k$ where x is an FDL constant or variable, and k is either a ground integer literal build from a natural number and possibly unary minus or a ground real literal involving one or more integer literal(s) and possibly unary minus and real division. 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.

`-push-down-to_real`

Rewrite applications of the integer-to-real conversion (`to_real` in SMT-LIB2) to integer arithmetic operators $\times, +, -(unary), -(binary)$, replacing the operators with their real counterparts and pushing the conversion down the term tree. Also rewrite applications of the conversion to integer literals to corresponding real literals.

`-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-non-const-real-div`

Abstract each real division unless the both argument are real constants. Needed by SMTLIB logics that support linear reals.

`-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.

`-abstract-to_real`

Abstract occurrences of the integer-to-real conversion operator to a new uninterpreted function. No characterising axioms are currently provided.

This option is needed to support Alt-Ergo v0.95 and before, since these provers do not recognise this operator.

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 CSV records, returning on standard output just those with *str* a substring of field *n* (1-based). If `-v` is provided, then returns 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. Fields are numbered starting from 1.

6.3 Project out fields of CSV records

Usage:

`csvproj` [`-v`] *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. If option `-v` is supplied, then all fields but *n1 ... nk* are used to build the output records. Fields are numbered starting from 1.

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 Missing Documentation

The current Victor has several features that are not properly documented yet in this manual. These include:

- 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.

8 Future developments

Here are some ideas for future next steps. Currently there are no definite plans for any of these to happen. If you are a Victor user and might be interested in any of these, please get in touch. Or, if you would like to take on a project to add any of these features yourself, please get in touch too.

1. **Improvement of support for proving VCs using interactive theorem provers.**

   The current Victor has some preliminary code for interfacing to Isabelle/HOL. Fabian Immler at TU München developed this further in 2010 and Secunet[2] took some interest in this interface. (Secunet since have developed their own SPARK FDL Isabelle/HOL interface.)

   Interfaces to other theorem provers such as PVS and HOL Light would be interesting to explore.

   Continued work on interfaces almost-certainly ought to exploit the incremental prover driver in order to write out a single prover input file for each SPARK unit.

2. **An API interface for Z3**. This could offer improved performance and increased functionality.

3. **A native input language Alt-Ergo interface.** This could offer improved performance and increased functionality, especially as the Alt-Ergo team currently appear to be emphasising support for its native input language. over SMT-LIB 1.2 or 2.

4. **Exploitation of Z3 smt-lib2 extensions for arrays and records.** Again, a benefit could be improved performance.

5. **Exploitation of the bit-vector handling capabilities of Smt solvers.** As a preprocessing step, this would involve recovering finite range information for integer-typed constants, functions and variables.

[2] http://www.secunet.com/

6. **Exploration of counter-example generation.** With the usual high number of quantified hypotheses and rules, SMT solvers virtually always report an *unknown* result rather than a *sat* result when proof fails.

   How could SMT solvers be persuaded to still return some tentative counter-example information, perhaps with some indication of which quantified assumptions were not fully examined and potentially could still refute the counter-example?

   The riposte project[3] has been exploring counter-example generation using answer-set programming. It would interesting to compare counter-examples produced by these two alternative methods.

[3] https://forge.open-do.org/projects/riposte/