Uppsala University
Department of Information Technology

The GTO formal methods tool

version 0.5.32

Reference manual

Table of contents

1 Introduction

The specification tool GTO is a simple tool for formal specification and verification work. GTO implements a restricted temporal many-sorted first-order predicate logic and supports simulation of and theorem proving in this logic whenever the sorts (types) of the language are assigned particular finite sets of objects.

This document provides a quick reference of GTO and its specification language, No attempt is made to explain a methodology for using GTO. This manual should be read and understood before attempting to use GTO for serious work.

The reader should be warned that GTO is a tool under development. This means that there may be partially implemented features which are not described in this reference. Also, in some cases complete and proper error checks may not be done.

There are no known errors in the program affecting the soundness of simulations or theorem proving, although there can be such errors which have not yet become manifest.

GTO was initially developed at Industrilogik L4i AB by Lars-Henrik Eriksson (except that the auxiliary program `dimacs` used to prepare input to theorem proving engines using the DIMACS format was written by Per Larsson). GTO has been extensively used in several commercial formal methods projects.

In 2005, Industrilogik was aquired by Prover Technology (www.prover.com) and that company now has the rights to the program. Continued development at Uppsala university is done by Lars-Henrik Eriksson (lhe@it.uu.se).

The name “GTO” refers to the Ferrari GTO racing car. In the right hands, the GTO tool is powerful, but if you are not careful, you will find yourself off the road.

The information in this document is current as of version 0.5.32 of GTO.

2 Operation

2.1 Installation

GTO is designed to run under Unix. Installing and running GTO requires a SICStus Prolog development system. GTO is tested with version 3.12.0 of SICStus, but it uses few "advanced" features and should run properly with considerably older versions of SICStus.

The installation scripts have been tried under SunOS 5.9 and Mac OS X 10.4. They should work under other Unix systems as well assuming that the Gnu utilities `gcc`, `bison` and `flex` are available. To use other compatible utilities, the makefiles must be changed.

To install GTO, place the GTO distribution files in a suitable directory (the GTO distribution directory) and give the shell command `make`. The executable file `gto.sav` will be created in the distribution directory and can be copied to any desired location and renamed as desired.

To remove files generated during the installation, give the shell command `make clean`.

2.2 Supported theorem proving engines

As of version 0.5.32, GTO supports SATO version 3.2, Z-Chaff versions 2001.2.17 and 2004.11.15 and Limmat version 1.3. Earlier versions of GTO also supported HeerHugo version 0.3 and CL-Tool version 5.0.5. GTO is likely to still work with these theorem provers, although that has not been tested. The theorem provers must be separately installed and available in the program search path when GTO is run. GTO also supports export of files in NP-Tools format.

Normally, Limmat is the default engine. The default engine can be changed by editing the file `defaultengine` in the GTO distribution directory.

2.3 Running GTO

Before GTO is run, the environment variable `GTODIR` must be set to the path of the GTO distribution directory. The path *must not* include a trailing slash. The SICStus development system as well as any theorem proving engines to be used must be in the program search path of the Unix shell used.

2.4 xgto

xgto is an experimental graphical user interface to GTO. xgto requires Tcl/Tk including the Expect library package. The Tcl/Tk shell `wish`, as well as GTO itself, must be in the default program search path. The environment variable `GTODIR` must be set as when running GTO directly. xgto has been tested with Tcl/Tk 8.4, but should work with version 8.1and up. Documentation for xgto is available on-line when running the program.

3 Language summary

3.1 Syntax descriptions

This section explains the various language constructs. The syntax of each construct is given using a line like:

`ALL` *v*[`:`*set*] *formula*

Characters which are to be entered verbatim are written in a `typewriter font`. Words in *italics* denote parts of the construct that can be different from instance to instance. Optional parts are enclosed in square brackets, [ ].

Repetition is denoted by an ellipsis (...), as in

`PRED` $pred_1$`(`s_1`,...,`s_n`),...,`$pred_n$`(...);`

In such cases it should be understood that there could be only a single instance of a repeated element. In that case no commas are used. In the case of repetition between parentheses, it should be understood that there could be no instances of the repeated element. In that case the parentheses are omitted. E.g.

```
PRED P;
```

for a declaration of a single predicate with no arguments (propositional variable).

3.2 Basic language elements

The various constructs of the language are built up from a number of basic elements:

Identifiers	Identifiers are used for variable and predicate names as well as for symbolic constants. Identifiers consist of letters (upper or lower case, including non-english letters of the ISO 8859-1 character set), digits and the underscore symbol (_). The first character in an identifier must be a letter. If the identifier is enclosed in single quotes (`'`), then arbitrary characters are allowed. To use a single quote character in a quoted identifier name, two consecutive quotes must be used. Example: The identifier written `'a.b'''` consists of four characters, `a`, a full stop, `b` and a single quote. Keywords of the GTO language (written with `CAPITAL LETTERS` in the syntax descriptions of sections 3.3 and 3.4) are not allowed as identifiers unless enclosed in single quotes.
Numbers	Numbers consist of sequences of digits. Only integers are allowed.

Strings	Strings are arbitrary sequences of characters enclosed in double quotes (`"`). To include a double quote in a string, it has to be written twice in succession.
Punctuation	Punctuation is parentheses, commas, semicolons and other characters used to impose a structure on language constructs.
Operators	Operators are characters other than those making up the previous constructs, such as `+` and `&`. **NOTE**: in some cases several characters can form one single operator such as `->`.
Comments	Comments are not properly language constructs, but are sequences of characters that are ignored by GTO. Comments are sequences beginning with `/*` and ending with `*/`. See also the command `comment`.

3.3 Statements

Statements of the language can be either a

- sort declaration
- constant declaration
- variable declaration
- predicate declaration
- input declaration
- output declaration
- predicate definition
- invariant
- file inclusion

Each statement must be terminated by a semicolon (`;`) character.

3.3.1 Sort declaration

Syntax:

`TYPES` $sort_1$[<$supsort_1$]`,` ... `,` $sort_n$[<$supsort_n$]`;`

Each $sort_i$ is declared to be a sort, i.e. a set to quantify over. If <$supsort_n$ is included, $sort_i$ is also declared to be a subsort of $supsort_n$.

The `TYPES` statement will also declare each $sort_i$ to be a one-argument predicate with an argument which can be of any sort. It will be implicitely defined to be true iff its argument is of the sort $sort_i$, i.e.

$sort_i$(X) == SOME y:$sort_i$ X=y;

Note that this is an entirely implicit definition – it can not be listed with the `listdef` command.

3.3.2 Constant declaration

Syntax:

CONST a_1, ... , a_n : $sort$;

Each a_i is declared to be a constant object of the given *sort*, i.e. a member of a set to quantify over. A constant can not belong to a several sorts (except implicitely in that it can belong to a sort which is a subsort of another sort).

3.3.3 Variable declaration

Syntax:

VAR v_1, ... , v_n : $sort$;

Each v_i is a variable name and *sort* is a sort name. The variables are declared to range over the objects in *sort*. It is permitted to omit the `VAR` keyword, although that syntax is deprecated and could be removed in coming versions of GTO.

3.3.4 Predicate declaration

Syntax:

PRED $pred_1$(s_1,...,s_n),...,$pred_n$(...);

Each $pred_i$ is declared to be a predicate. All predicates used must be declared before their first use. The s_1,..,s_n declare the sorts of the predicate arguments. Each s_i can be either a sort name or a declared variable. In the latter case, the declared sort of the variable is used as the corresponding argument sort.

3.3.5 Input declaration

Syntax:

INPUT $pred_1$,...,$pred_n$;

Each $pred_i$ is declared to be an input predicate for the purpose of the simulator.

3.3.6 Output declaration

Syntax:

`OUTPUT` $pred_1$`,...,`$pred_n$`;`

Each $pred_i$ is declared to be an output predicate for the purpose of limiting output from GTO (see section 4.3.22).

3.3.7 Predicate definitions

There are two different forms of predicate definitions. Only one of the forms can be used with each particular predicate.

3.3.7.1 Form 1

Syntax:

$pred$`(`v_1`,...,`v_n`)` `==` $formula$`;`

The predicate *pred* with arguments v_1`,...,`v_n is defined to be the given *formula*. The argument variables must be declared by a variable declaration statement. v_1`,...,`v_n are considered as bound variables in *formula*.

These definitions are not necessarily definitions in the logical sense as in two cases the predicate being defined may occur in the formula or in other definitions on which the formula depends.

- The definition may depend on the predicate being defined if it is within the scope of the `PRE` operator.
- *pred* may occur in *formula*, i.e. the definition may be immedeately self-recursive, although this is to used only when necessary as GTO in some cases handles such definitions incorrectly (see section 5.4.2). In this case a warning is given.

From a logical point of view, predicate definitions are more properly seen as equivalences. They do have a particular operational meaning for the simulator which motivates the special syntax (see section 5.4.1).

3.3.7.2 Form 2

Syntax:

`FACTS` $pred_1$`(`a_1`,...,`a_n`),...,`$pred_n$`(...);`

Each predicate instance $pred_1$`(`a_1`,...,`a_n`)` is defined to be true. The a_1`,..,`a_n are constant symbols of the appropriate sorts. Once a predicate has occurred in a `FACTS` statement,

GTO will assume that the complete set of true instan ces is listed, i.e. every instance not listed is implicitely defined to be false.

Note that it is not possible to define a predicate which is always false using this form of predicate definition.

3.3.8 Invariants

Syntax:

formula`;`

The *formula* should always be true, i.e. be an axiom

3.3.9 File inclusion

There are two different forms of file inclusion.

3.3.9.1 Form 1

Syntax:

`USE` *filename*`;`

The contents of the specification file with name *filename* is included at this point. The *filename* should adhere to the syntax for identifiers according to section 3.2. If this is not possible, e.g. if the file name includes a full stop character (.), the file name should be enclosed in single or double quotes – either is acceptable.

If no file with the given name can be found, the inclusion is retried with the chartacters ".gto" appended to the file name. Thus if the file name ends with ".gto", the ending can be omitted. If *filename* is not an absolute file path, the file will be looked up relative to the directory of the file containing the `USE` statement.

3.3.9.2 Form 2

Syntax:

`REFINES` *filename*`;`

This form is equivalent to form 1 except that 1) invariants in the included file will not be assumed to be true when doing theorem proving (see section 5.6) and that 2) `INPUT` declarations in the included file will be ignored.

3.4 Formulae

Formulae are constructed from logical connectives, predicates and relations in the following manner:

`~`*formula*	Negation (not)
`PRE` *formula*	Previous moment
`ALL` *v*[`:`*sort*] *formula*	Universal quantification (for all)
`SOME` *v*[`:`*sort*] *formula*	Existential quantification (for some)
formula `&` *formula*	Conjunction (and)
formula `#` *formula*	Disjunction (or)
formula `->` *formula*	Implication (if ... then ...)
formula `<->` *formula*	Equivalence (if and only if)
`TRUE`	Constant truth
`FALSE`	Constant falsity
expression `=` *expression*	Equality
expression `<>` *expression*	Inequality
pred `(` *expression* `,` ... `,` *expression* `)`	Predicate instance

If the *sort* is omitted from a quantified formulae, the bound variable *v* must be declared by a variable declaration statement, and the sort to quantify over is taken from this declaration.

In the present version of GTO, the `PRE` operator may not be nested except in formulae used as arguments to the `prove` and `satisfy` commands.

Expressions are constructed in the following manner:

identifier	Bound variable or symbolic constant

3.5 Semantics

The logic of the GTO tool is a many-sorted temporal logic with linear time. It is further restricted in that no function symbols are used and the only a single temporal operator – `PRE` – is used (note that the `PRE` opertor may generally not be nested in the present version of GTO). Sorts may form a hierarchy by set inclusion.

Although not inherent in the logic, GTO assumes that the contents of all sorts are given and are finite, so that the formulae can be rewritten without quantifiers. For this reason, the semantics is defined relative to a particular sort contents.

The semantics of a formula is given relative to a “time line” T and a “sort assignment” S. The time line T is a set indexed by the natural numbers. Each element of the time line – denoted T_i – is a “model” in the traditional sense, i.e. a set of true predicate instances. The sort assignment S is a set indexed by sort names. Each element of S – denoted S_s – is the set of constants belonging to the sort s. It is assumed that S respects the subsort hierarchy,

i.e. that if s is a subsort of t, then $S_s \subseteq S_t$. Also, if $S_s \cap S_t \neq \emptyset$, then either s=t, s is a subsort of t or t is a subsort of s. Sorts may be empty.

The satisfiability of a formula F relative to a given time line T and sort assignment S at a given time instance i>0 – written $T,i \models_S F$ – is defined inductively over the structure of F as follows:

$T,i \models_S$ `~`F	iff not $T,i \models_S F$
$T,i \models_S$ `PRE` F	iff $T,i-1 \models_S F$
$T,i \models_S$ `ALL` v`:`s F	iff for every $a \in S_s$, $T,i \models_S F[a/v]$ (vacously true if $S_s=\emptyset$)
$T,i \models_S$ `SOME` v`:`s F	iff for some $a \in S_s$, $T,i \models_S F[a/v]$ (vacously false if $S_s=\emptyset$)
$T,i \models_S F$`&`G	iff both $T,i \models_S F$ and $T,i \models_S G$
$T,i \models_S F$`#`G	iff either $T,i \models_S F$ or $T,i \models_S G$
$T,i \models_S F$`->`G	iff either $T,i \models_S G$ or not $T,i \models_S F$
$T,i \models_S F$`<->`G	iff both or neither $T,i \models_S F$ and/nor $T,i \models_S G$
$T,i \models_S$ `TRUE`	always holds
$T,i \models_S$ `FALSE`	never holds
$T,i \models_S x$`=`y	iff x and y are the same object
$T,i \models_S x$`<>`y	iff x and y are different objects
$T,i \models_S pred$	iff $pred \in T_i$ (*pred* is a predicate instance)
$T,i \models_S$ `INV` F	iff for every j, $0<j \leq i$, $T,j \models_S F$

Note that the temporal operator `INV` is not currently a part of the GTO formula language. It is used to define the behaviour of the `prove` and `satisfy` commands (see section 5.6).

A formula F is a **valid** relative to a sort assignment S – written $\models_S F$ – iff for every T and i $T,i \models_S F$ holds.

A formula F is a **logical consequence** of a set of formulae Γ, relative to a sort assignment S – written $\Gamma \models_S F$ – iff for every T and i such that for all $G \in \Gamma$, $T,i \models_S G$ it is the case that $T,i \models_S F$ also holds.

Corollary: F`->`G is valid iff G is a logical consequence of F.

4 Command summary

4.1 GTO command level

GTO reads commands interactively from the standard input. When ready to accept a command, it will display a prompt character:

```
>
```

After processing a command and presenting the output from the command, GTO will read another command until reaching end of file on the standard input or the `quit` command is given.

Commands read from standard input are terminated by a newline character (typically using the RETURN or ENTER key on the keyboard). If a command is long, it may be broken up into several lines. When a newline is entered, GTO will recognise that additional input is needed to complete the command. A continuation prompt

```
...
```

is displayed and the user can enter the next part of the command. For example:

```
> evf p &
... q
```

The command will be acted upon when sufficient input has been given to complete the command. Of course, this only works if GTO can recognise that the first line of input is incomplete. If the command line was broken between the `p` and the ampersand sign, GTO would process the command directly rather than display a continuation prompt as `evf p` is a valid command in itself.

4.2 Syntax descriptions

This following section will explain GTO commands. The syntax of each command is given using a line like:

`pef` *formula*

Characters which are to be entered verbatim are written in a `typewriter font`. Words in *italics* denote parameters to the command. Optional parameters are enclosed in square brackets, [ ].

In all cases where the command description states that a logical formula is used, it is possible to enter an *invariant identifier* instead. Invariants in specification files are identified with the name of the file, followed by an underscore and a sequence number, e.g.

```
speca_3
```

identifies the third invariant in the specification file `speca`. The use of an invariant identifier will behave as if the identifier was a predicate defined by the invariant formula.

Parameters which are *file names* should adhere to the syntax for identifiers according to section 3.2. If this is not possible, e.g. if the file name includes a full stop character (.), the file name should be enclosed in single or double quotes – either is acceptable.

4.3 Command descriptions

4.3.1 comment (do nothing)

Syntax:

`comment` *anything*

The comment command does not do anything. Its is intended to be used in command files to provide comments that will be written to the standart output for logging purposes. Ordinary comments (sequences beginning with `/*` and ending with `*/`) will be discarded when the command file is being read and so never echoed on the standard output.

anything is a (possibly empty) sequence of arbitrary language elements.

4.3.2 depends (print predicate dependencies)

Syntax:

`depends` $pred_1$ [$pred_2$]

If $pred_2$ is omitted, the `depends` command prints the names of all dynamic (non-static) predicates on which the definition of the dynamic predicate with name $pred_1$ directly or indirectly depends. The listing will be broken down in different categories of dependencies (currently dependencies on defined and input predicates at the present moment of time, dependencies on a predicate at the previous moment of time and finally dependencies on predicates which are neither defined nor input predicates).

If $pred_2$ is given, the command will write `Yes` or `No` depending on whether $pred_1$ depends on $pred_2$ or not. If the dependency is to the previous moment of time, `Yes (PRE)` will be written.

4.3.3 do (perform a simulation step)

Syntax:

`do` [[~]*pred*`(`*expression*`,...,`*expression*`)` … [~]*pred*`(`*expression*`,...,`*expression*`)`]

A simulation step is carried out with the given input predicate instances set true (or false, if prefixed with `~`) at the new moment of time (see section 5.4). Input predicate instances not mentioned will keep their truth values from the previous moment. In particular, if no predicate instances are given, the simulation step is carried out with all input predicate values unchanged.

All predicate instances (except for the input predicates) which are changed by the simulation step are listed. If the `nowriteall` command has been given, the listing is limited to output predicates.

The identifiers of any invariants that are violated by the simulation step are listed.

4.3.4 evf (evaluate formula)

Syntax:

`evf` *formula*

The truth value of *formula* in the current simulator state is printed.

4.3.5 engine (choose theorem proving engine)

Syntax:

`engine` [*name*]

If *name* is given, the theorem proving engine identified by *name* is chosen for use by the `export`, `satisfy` and `prove` commands. If *name* is not given, the command displays the name of the currently chosen theorem proving engine

As of version 0.5.32 of GTO, *name* can be one of `heerhugo`, `nptools`, `zchaff`, `limmat`, `cltool` or `sato`.

4.3.6 export (export specification)

Syntax:

`export` *filename*

The loaded specification is partially evaluated (see section 5.5) and written to a file with the given name in a format suitable as input to the chosen propositional theorem prover engine. Identifiers are rewritten as necessary to adhere to the syntax of the chosen engine. In particular, if the chosen engine uses the DIMACS input format, GTO identifiers will be rewritten to numerical identifiers with no connection to the originals.

4.3.7 help (give help with commands)

Syntax:

```
help
```

A help message is printed, summarising the available GTO commands.

4.3.8 info (give predicate information)

Syntax:

`info` [*pred*]

Without argument, GTO lists which predicates are in each of the following classes:

- Input predicates (predicates declared in an INPUT statement)
- Output predicates (predicates declared in an OUTPUT statement)
- FACT predicates (predicates defined using form 2 of the definition statement)
- Static predicates (predicates which are known to have the same values at every time instance, see section 5.1).
- Dynamic predicates (predicates which are not static)
- Completed predicates (predicates for which GTO has constructed definitions from invariants, see section 5.3)

With argument, GTO writes which of the above categories the predicate with name *pred* is in.

4.3.9 init (initialise the simulator)

Syntax:

```
init
```

The simulator is initialised, all dynamic predicates instances are set to the truth value false and one simulation step is made (see section 5.4). To begin simulating with other initial values, use the `satisfy` command with a formula describing the desired initial values.

All predicate instances which are changed from their initial values by the simulation step are listed. If the `nowriteall` command has been given, the listing is limited to output predicates.

The identifiers of any invariants that are violated are listed.

4.3.10 list (list predicate values)

Syntax:

`list` [*n*] [$pred_1$ ... $pred_n$]

GTO prints all true instances from the current simulator state of the predicates with names $pred_1$ through $pred_n$. If *n*=0 only the true instances of static predicates are printed. If *n*>0, true instances from n time steps back including the present are printed. If *n* is omitted, 1 is assumed.

The number of steps back in time that can be listed depends on the how the simulator state was generated. After a simulation step, the maximum value of *n* is 2. After generating a model with the `satisfy` command or a countermodel with the `prove` command, the maximum value of *n* is equal to the time window settting.

If no predicate names are given, the true instances of all predicates are printed (however if *n* is given and greater than zero, only instances of non-static predicates are printed). If the `nowriteall` command has been given, the listing is limited to all output predicates.

Predicate instances are listed in two groups, first the dynamic predicate instances, then the static predicate instances. Within each group, the instances are sorted by predicate name.

4.3.11 listdef (list definitions)

Syntax:

`listdef` [*pred*]

GTO prints the definition of the predicate with name *pred*,provided is has been defined using definition form 1. If *pred* is omitted, all such predicate definitions are printed. The listing includes the definitions of completed predicates, which are constructed by GTO itself.

4.3.12 listinv (list invariants)

Syntax:

`listinv` [*inv*]

GTO prints the invariant with identifier *inv*. If *inv* is omitted, all invariants are printed.

4.3.13 load (load specification)

Syntax:

`load` *filename*

GTO clears its internal database and loads a new specification from the given file. Any previously loaded specification is completely replaced.

If no file with the given name can be found, the inclusion is retried with the characters ".gto" appended to the file name. Thus if the file name ends with ".gto", the ending can be omitted. If *filename* is not an absolute file path, and the `load` command is given from within a command file, the file to load will be looked up relative to the directory of the command file.

When the specification has been loaded, GTO performs various checks and initialisations, including

- checking the truth value of invariants which can be determined to always have the same truth value in every time instance.
- completing definitions of predicates.

4.3.14 pef (partially evaluate formula)

Syntax:

`pef` *formula*

The given formula is partially evaluated (see section 5.5) with respect to the static predicates and the `CONST` declarations. If the formula is a single predicate instance of a defined predicate (defined explicitely by a form 1 definition or by completion), the definition formula will be partially evaluated instead, taking into account the particular arguments given to the predicate.

4.3.15 prove (prove a formula)

Syntax:

`prove` *formula*

GTO attempts to show that the given formula is a logical consequence of the definitions and invariants of the loaded specification (except invariants in files included using `REFINES`).

If the formula is found to be falsifiable, the current simulator state will be set to some countermodel of the formula.

Nested `PRE` operators **are** allowed in *formula*.

IMPORTANT: The `prove` command may be incomplete and report false countermodels. See section 5.6 for a full understanding of the command.

4.3.16 pulse (simulation with temporary input)

Syntax:

`pulse` [[~]*pred*`(`*expression*`,...,`*expression*`)` … [~]*pred*`(`*expression*`,...,`*expression*`)`]

This command is equivalent to the `do` command, except that after the simulation step, the given input predicate instances are reset to the opposite value and another simulation step is done.

4.3.17 quit (quit GTO)

Syntax:

quit

GTO exits.

4.3.18 satisfy (generate a model)

Syntax:

`satisfy` *formula*

GTO attempts to show that the given formula is satisfiable in a model also satisfying the definitions and invariants of the loaded specification (except invariants in files included using `REFINES`).

If the formula is found to be satisfiable, the current simulator state will be set to some model of the formula.

Nested `PRE` operators **are** allowed in *formula*.

IMPORTANT: The `satisfy` command may be unsound and report false models. See section 5.6 for a full understanding of the command.

4.3.19 take (take commands from a file)

Syntax:

`take` *filename*

GTO starts reading commands from the given file. When the entire file has been read, it reverts to reading commands from its standard input. Command execution from files proceeds in the same way as when commands are entered from the standard input with two exceptions:

- Commands taken from a file are not limited to single lines (or sequences of lines). Every command must be terminated by a semicolon (;) character. Several commands may be put on a single line and commands may be broken into several lines at any point. The automatic command line continuation feature described in section 4.1 is not

applied.
- Each command taken from a file is written to the standard output before execution (preceded by the prompt character >) to make it easier to read the output.

Should an error occur while commands are taken from a file, reading of the file will be aborted and input reverts to the terminal.

If no file with the given name can be found, the command is retried with the characters ".gto" appended to the file name. Thus if the file name ends with ".gto", the ending can be omitted. If *filename* is not an absolute file path, and the `take` command is given from within a command file, the new file to read commands from will be looked up relative to the directory of the current command file.

Command files can not be nested. A `take` command in a command file will cause GTO to read commands from the new command file, but when the new file ends, it will not revert to the previous command file.

Example: `take 'commands2'`

4.3.20 timewindow (set/display time window)

Syntax:

`timewindow [`*n*`]`

If *n* is given, the time window for theorem proving (see section 5.6) is set to that number. If *n* is omitted, the current value of the time window is printed. The initial value of the time window is 1.

4.3.21 why (find witnesses to formulae)

Syntax:

`why` [*formula*]

GTO attempts to find *witnesses* to the truth or falsity of the given formula in the current simulator state. If formula is omitted, the command will use the argument of the previous `prove`/`satisfy` command or the formula given as the final explanation of the previous `why` command, depending on which of these commands where used most recently.

If *formula* is an atomic instance of a predicate defined using definition form 1(see section 3.3.7), the definition will be applied to the instance and witness to the definiens will be sought.

A witness to an existentially quantified formula `SOME` *x P* is an expression such that P would become true if *x* was replaced by the witness. Similarly, a witness to a universally quantified formula is an expression which makes the formula body false.

A witness to a disjunction or conjunction is (for the purpose of this command) a subformula which makes the disjunction true or false, respectively.

A witness to a negation is the same as the witness of the negated formula.

A witness to an implication is the same as the witness of the implied formula.

If `why` is unable to find a witness, it will partially evaluate the formula and try again. If it still cannot find a witness, it will reply `Don't know`.

4.3.22 writeall (set/display output restriction)

Syntax:

`writeall [`*flag*`]`

If *flag* is `yes`, printing of predicate values by the commands `do`, `init`, `list` and `pulse` will be limited to values of output predicates. If flag is `no`, no such limitation is applied. If *flag* is omitted, the current value of the writeall flag is printed. The initial value of the flag is `yes`.

5 Processing the specification

5.1 Collecting information from the specification file

When a specification file is loaded, various processing and checks are made. The information in the loaded file (and included files) are used to define the following sets, which are used by the simulator and theorem prover:

S	The sort assignment.
Δ	The set of definitions.
P_s	The set of static predicates.
Δ_s	The set of static definitions.
Ξ	The set of invariants.
Ξ_s	The set of static invariants.
Ξ_r	The set of refined invariants.
T_0	The initial model.

The sort assignment, S, is constructed using the `CONST` and `TYPES` statements. The index set of S comprises every sort declared in a `TYPES` statement. Each S_s will include exactly the following constants:

- Every a_i from every statement of the form `CONST` a_1`,` ... `,` a_n `:` s`;`
- Every $a \in S_t$, for every subsort declaration `TYPES` ...`,`s`<`t`,`...`;`

The set of definitions, Δ, is the set of formulae of the form

`INV ALL` v_1 ... `ALL` v_n (*pred*`(`v_1`,`...`,`v_n`)` `<->` F)

obtained from every form 1 definition

pred`(`v_1`,`...`,`v_n`)` `==` F

as well as from every form 2 definition reexpressed as a form 1 definitions.

The set of static predicates, P_s, is the least set satisfying the following condition:

- *pred*$\in P_s$ iff *pred* is a defined predicate and the every predicate occurring in the definition is in P_s.

The values of the static predicates are independent of the particular time instance.

The set of static definitions, Δ_s, is the subset of Δ which includes exactly the definitions of the static predicates, including the predicates corresponding to sort names (see section 3.3.1).

The set of invariants, Ξ, is the set of formulae of the form

```
INV F
```

obtained from every invariant, *F*, in the specification.

The set of static invariants, Ξ_s, is the subset of Ξ which includes exactly those invariant formulae which only depend on static predicates.

The set of refined invariants, Ξ_r, is the subset of Ξ which includes exactly those invariants occuring in subfiles included using the `REFINES` statement.

The initial model, T_0, is always set to the empty set in version 0.5.32 of GTO. In future versions, it will be possible to set the contents of T_0.

5.2 Initial processing of the specification file

After the information has been collected, GTO computes the values of all the static predicates and checks that all the static invariants are true. An error message is given if any static invariant is false.

5.3 Completion

After loading the specification, GTO attempts to complete every predicate which does not have a definition and is also not an input predicate. This involves the construction of a definition for every such predicate for the benefit of the simulator (see section 5.4).

Let *pred* be a predicate to be completed. The completion procedure works by using information in invariants involving *pred* in question. The completion procedure will look for invariants of the following form:

- `ALL` v_1 ... `ALL` v_n `(`*pred*`(`v_1`,...,`v_n`) <->` F`)`
- `ALL` v_1 ... `ALL` v_n `(`F `<->` *pred*`(`v_1`,...,`v_n`))`
- `ALL` v_1 ... `ALL` v_n `(`*pred*`(`v_1`,...,`v_n`) ->` F`)`
- `ALL` v_1 ... `ALL` v_n `(`F `->` *pred*`(`v_1`,...,`v_n`))`

If there are invariants of the first form, one of them is used to construct a definition

pred`(`v_1`,...,`v_n`) ==` F

If there are no invariants of the first form, but some of the second form, one of these is used to construct a definition of the same form.

If there are no invariants of the first or second forms, but some of the third form, the *F* from each such invariant is collected. If necessary, the quantified variables are renamed so

the same variables are used. Call the collected formulae $F_1,\ldots,F_n$. The following definition is then constructed:

$$pred(v_1,\ldots,v_n) \;\; == F_1\&\ldots\&F_n$$

If there are no invariants of either the first, second or third forms, but some of the fourth form, the F from each such invariant is collected. If necessary, the quantified variables are renamed so the same variables are used. Call the collected formulae $F_1,\ldots,F_n$. The following definition is then constructed:

$$pred(v_1,\ldots,v_n) \;\; == F_1\#\ldots\#F_n$$

By substituting the constructed definition in the invariant(s) used to construct it, it is easily seen that the definition can not falsify the invariant(s). Thus it is "safe" to use the completed definitions. This is does not mean, of course, that the completed definitions are sensible from the user's point of view. The user must ensure that the completions are sensible and if they are not write explicit definitions for the benefit of the simulator.

The `listdef` command can be used to view the constructed definitions of completed predicates.

The names of predicates that could not be completed is output to the user.

5.4 Simulation

5.4.1 The function of the simulator

The purpose of the simulator is to successively construct a time line T such that every formula in Δ and Ξ is valid relative to S (if possible).

The simulator achieves this using the initial model T_0 and computation rules to compute models T_i at later times $i>0$, given T_j for each $j<i$. Given the present restriction to the language that `PRE` operators may not be nested, it suffices to consider $j=i-1$ (however, see the comment about invariants below).

The computation rule is determined as follows:

- For input predicates, the value of the predicate is obtained from the user. If the user does not provide a value, the same value as in T_{i-1} is used.
- For defined predicates, the value of the predicate is computed from the truth value of its definition.
- For completed predicates, the value of the predicate is computed from the definition constructed using the completion procedure described in section 5.3.

If the computation rule does not cover every predicate (i.e. if the completion procedure failed for some predicate), simulation can not be done.

At each moment, the simulator keeps a particular T_i – the "current state" of the simulator. When a simulation step is carried out as a result of giving the `do` or `pulse` commands, the time instance is incremented by one, so the current state becomes T_{i-1}. The computation rules are then used to compute the new current state, after which the old state is no longer needed. In fact the value of i is irrelevant, so the simulator considers only the "old state" (T_{i-1}) and the "new state" (T_i) during the computation step.

When the simulator is initialised using the `init` command, the current state is set to an initial state where all dynamic predicate instances are false. A simulation step is then carried out as described above.

With the exception mentioned below, the computation rules are such that each formula in Δ is guaranteed to be valid, the same holds for the formulae in Ξ used to generate the completing defininitions. In general, however, the simulator must verify at each step that invariant formulae in Ξ hold. This is done by computing their truth value and displaying an error message if any invariant should have a false truth value. Since the static invariants have been checked once and for all when the specification was loaded, the simulator will actually only check the non-static invariants (i.e. the invariants in $\Xi \backslash \Xi_s$).

To be strict, the formulae in Ξ are all of the form `INV` F, so once an F evaluates to false in a particular simulation step (or even in the initial state), the formula `INV` F should remain false indefinitely. To the user of the simulator, it is more convenient to have the original invariant formula F reevaluated at every simulation step, so this is what the simulator actually does.

5.4.2 Self-recursive definitions

Definition may be self-recursive, i.e. the predicate being defined may occur in its own definition without being inside the scope of a `PRE` operator. Such definitions may be inconsistent or ambigous. Be warned that GTO does **not** make any check that the definition is consistent and unambigous. The user is provided with a warning that the definition is self-recursive, but the task of ensuring that the definition makes sense is up to the user.

For every self-recursive predicate, the computation of truth values of self-recursive predicates is carried out by repeatedly (re)computing the predicate according to the computation rule until a fixpoint is found. There is in general no guarantee that a fixpoint will be found (in particular if none exists e.g. if the definition is inconsistent) and in such cases the computation will not terminate, i.e. GTO will hang. Also, if there are multiple fixpoints the particular fixpoint found is undetermined.

The recomputation of self-recursive predicate can potentially be very time-consuming. To keep down the number of recomputations required, the order in which GTO computes different predicate instances should be kept in mind. Each time a self-recursive predicate is (re)computed, the truth values of the predicate instances are computed in a particular order determined by of the arguments of the predicate. The order on argument tuples $(a_1, \ldots, a_n) < (b_1, \ldots, b_n)$ is a left-right lexicographic order generated from the order in

which the constants involved are declared in the specification file. For constants *a* and *b*, *a*<*b* iff *a* is declared ahead of *b* in the file.

This means that if the definition of a self-recursive predicate *pred* is such that the truth value of each predicate instance *pred*`(`b_1`,...,`b_n`)` depends only on the truth values of predicate instances *pred*`(`a_1`,...,`a_n`)` where `(`a_1`,...,`a_n`)` < `(`b_1`,...,`b_n`)`, then *pred* can be computed with no recomputations. The degree in which a predicate instance depends on instances later in the argument order determines the number of recomputations necessary.

5.5 Partial evaluation

Partial evaluation is a process where a formula is simplified using partial knowledge about the values of the components making up the formula. In the case of the GTO tool, this information is primarily the values of the static predicates and the elements of the sorts.

In the GTO tool, partial evaluation is carried out in two conceptually distinct steps.

1) Quantifiers are removed:

- `ALL` *v*`:`*sort* *formula* is replaced by *formula*[a_1/*v*]`&`…`&`*formula*[a_n/*v*], where *sort* = {a_1,…,a_n}. The empty conjunction is taken as `TRUE`.
- `SOME` *v*`:`*sort* *formula* is replaced by *formula*[a_1/*v*]`#`…`#`*formula*[a_n/*v*], where *sort* = {a_1,…,a_n}. The empty disjunction is taken as `FALSE`.

2) Instances of static predicates are replaced by their values. (This is always possible as all variables have been removed by step 1.)

3) Some simplifications are made using basic logical laws – particularly laws involving `TRUE` and `FALSE`. E.g. `TRUE &` *F* ≡ *F*, and *F* –> *F* ≡ `TRUE`.

A partially evaluated formula *F*' will be equivalent to the original formula *F* assuming the static definitions, i.e.:

$$\Delta_s \models_S (F' \leftrightarrow F)$$

5.6 Theorem proving

GTO can interface to an external propositional logic theorem prover, also known as a satisfiability (SAT) solver. Using the commands `prove` and `satisfy`, theorem proving problems can be generated and any resulting (counter)models returned to GTO.

When the command `prove` *formula* is given, GTO attempts to show that the formula is a logical consequence of the definitions and non-refined invariants, i.e. that

$$\Delta, \Xi \backslash \Xi_s \models_S formula$$

If this relation does not hold, there must be some time line T and some time instance i such that the definitions and invariants are satisfied but the formula is falsified. The current simulator state will be set to T_i. The choice of T and i is arbitrary.

When the command `satisfy` *formula* is given, GTO attempts to find a time line T and time instance i such that all of the following hold:

$T,i \models_S formula$
$T,i \models_S D$, for each $D \in \Delta$
$T,i \models_S G$, for each $G \in \Xi \backslash \Xi_s$

If some such T and i are found, the current simulator state will be set to T_i.

In both cases above, GTO must create a propositional theorem proving (satisfiability) problem. It achieves this by first partially evaluating *formula* and all elements of Δ and $\Xi \backslash \Xi_s$. The resulting formulae will in propositional form (quantifier and variable-free), but any temporal operators remain.

The `INV` operator is eliminated by rewriting, observing that according to the semantics:

$T,i \models_S$ `INV` F iff $T,i \models_S F$ `&(PRE` F`) &(PRE PRE` F`) &...& (PRE PRE ... PRE` F`)`

where the number of conjuncts is equal to i+1. Since this can be arbitrary large for arbitrary large i, only a fixed number of conjuncts with the least number of `PRE` operators are included in the rewriting. The number of conjuncts included is referred to as the time window, and can be set using the `timewindow` command (see section 4.3.20).

WARNING: From the preceding paragraph it should be clear that the translation to propositional logic is not entirely correct since it does not fully respect the semantics of `INV`. This causes the `prove` command to be incomplete and it may report false counterexamples. Likewise, the `satisfy` command will be unsound and may report false models. This shortcoming can be rectified by applying a suitable methodology using induction when carrying out proofs with GTO. However, a description methodologies is presently outside the scope of this manual.

The `PRE` operator is eliminated by renaming the predicates within the scope of the operator so that the operator is implicit in the predicate name. Finally the predicate instances are syntactically changed to propositional variables. The result will be entirely in propositional logic and can be used to generate appropriate problems for the propositional solver.

Should a counterexample be generated, GTO translates the truth value assignments to propositional variables back into truth value assignments to the predicate instances at the proper place in the timeline.