# The GTO Toolset and Method

Lars-Henrik Eriksson

# The GTO Toolset and Method

Lars-Henrik Eriksson[1,2]

*Department of Information Technology*
*Uppsala University*
*Box 337*
*SE-751 05 UPPSALA, Sweden*

**Abstract**

A suitable method supported by a toolset with a high degree of automation is a necessity for the successful employment of formal methods in industrial projects. The GTO toolset and method have been developed, and successfully applied, to formal methods in safety-critical control applications related to railway signalling since the mid 1990s.
The toolset and method support the entire formal methods process from writing and validating formal specifications, through modelling of the implementation to formal verification and analysis of verification results. One goal the toolset and method was to make formal methods more competitive by streamlining the process so that – at least within an established application area – individual verification tasks could be done in an "assembly line"-like fashion with minimum overhead.
In line with this goal, the toolset is intended for use with configurable systems, where a generic specification is applicable to a family of systems and adapted to a specific system using configuration data.
The functions carried out by the toolset include static checking and simulation of specifications, checking of configuration data, generation of implementation models from PLC program code or relay schematics, simulation of the implementation model, formal verification by refinement proof, and analysis of failed refinement proofs. Refinement proofs are automatically carried out by a satisfiability (SAT) solver of the user's choice, which is interfaced to the main tool.
We will outline the method and functions of the toolset as well as the formal notation – a simple temporal predicate logic – used by the toolset.

*Keywords:* Formal methods process, Formal methods tools, Formal verification, Generic specification, Refinement proof

## 1 Introduction and background

Formal methods are increasingly making their way into development and quality assessment of safety-critical systems. In particular, the development during the last decade of very fast propositional satisfiability (SAT) solvers has made automatic verification of real-size industrial systems possible.

For the past 10 years the consultancy Industrilogik developed and used a toolset and method for formal verification of control systems – in particular for railway signalling. Early projects, while technically successful, required much project-specific work by formal methods experts – they had an experimental character with high

[1] This work was done while the author was at Industrilogik L4i AB (in 2005 acquired by Prover Technology AB). I wish to thank my former colleagues for their involvement.

[2] Email: lhe@it.uu.se

*A revised version of this paper will be electronically published in*
*Electronic Notes in Theoretical Computer Science*
*URL:* www.elsevier.nl/locate/entcs

cost. One explicit goal of the company was to verify new systems in an "assembly-line" fashion with minimum overhead – a goal which has been realised to a major extent. The success is the result of several factors:

- The adoption of an established and well-founded theoretical basis — synchronous modelling [1] and linear time temporal logic [4].
- A simple formal notation that is easy to paraphrase in natural language when communicating with clients and application experts.
- The development of formal specifications which are *generic*, i.e. formulated in general terms and not by reference to any particular system. An example is a specification describing general railway signalling principles rather than the requirements of a particular installation. Using generic specifications, the specification work can for the most part be reused from one verification project to the next within the same application area.
- Tools which permit largely automatic translation of the implementation from common description formats into the notation used by the verifier.
- A verification tool which interfaces to any state-of-the-art SAT solver and permits the user to analyse the results (counter-examples) of failed verifications in terms of concepts used in the specification.
- The development of methods and strategies for the application area.

Here I will give an overview of the method, notation and toolset — in particular the verification tool GTO [3]. While most techniques used in and together with the tool are well established today, the particular combination was innovative when GTO was developed and has proven successful in many industrial projects. Notable projects include the analysis of the signalling system involved in the Åsta railway collision in Norway in 2000 [5] and the ALISTER interlocking development for the Swedish National Rail Administration.

The initial inspiration for the GTO tool was the experimental modelling tool "Delphi" developed by Ericsson (the telecom company) together with Prover Technology AB [4] in the early 1990s. The tool used a graphical modelling notation combined with predicate logic and an event-based method for describing state changes as truth maintenance problems which was innovative but computationally complex. The models were analysed by the Stålmarck SAT solver [12], after translating them into propositional logic by fixing the sizes and contents of the involved sets. In many ways the objectives and ideas behind this modelling tool were similar to those behind the Alloy tool [10].

GTO was a complete re-implementation in Prolog done by the author at Industrilogik L4i AB, shifting the focus from modelling to verification. The basic idea of having a predicate logic notation which was translated into propositional logic was kept, but the truth maintenance function was discarded. The event notion of state change was later replaced by a proper (albeit simple) temporal logic. Also, the tool was made independent of any particular SAT solver by introducing a general inter-

[3] Named after the Ferrari GTO racing car. A remark about an early version of the tool was that like the car, it can take you places quickly, but you must be skilled to handle it.

[4] At the time called Logikkonsult NP AB.

face. The tool has been used with HeerHugo [7], Sato [14], Z-Chaff [11], Limmat [2] and a further development of the Stålmarck solver, the Prover CL-Tool. There is also an experimental interface to the NuSMV symbolic model checker [3].

Symbolic model checkers are extensively used today used to solve verification problems. However our experience is that SAT solving appears to work much better than model checking for the class of problems encountered in railway signalling. Reports of the use of model checkers in this domain [9] as well as comparative experiments we have made support this impression.

## 2 Overview

This section gives an overview of the notation, tools and method. They are illustrated by a complete example in section 3.

### *2.1 Basic Principles*

The method is based on modelling of specifications and implementations in a notation based on temporal logic. States of the implementation and its interfaces to the environment are represented by truth values of predicates which can change over time, so the behaviour of the system is described by a sequence of truth value assignments. The concept of time is discrete and linear. Following the synchronous hypothesis [1], the system is assumed to change state faster than its environment, so actions can be assumed to be instantaneous and the time steps infinitesimally short.

As the requirements specification model determines the permissible behaviour of the system, while the implementation model determines the possible behaviours of the system, a correct implementation must be a refinement of the specification. The refinement relation is established by showing that the invariant formulae of the specification model are logical consequences of (i.e. can be proved from) the implementation model formulae (together with any definition formulae of the specification model).

### *2.2 Notation*

The formal notation of GTO is a variant of LTL (linear time logic)[4], using past-time modalities. The logic is extended to include many-sorted predicate logic, i.e. quantified formulae with different quantification domains and predicates. There are several restrictions to the logic intended to make the implementation simpler and more efficient. In particular, all sorts must be finite (the values of each sort explicitely enumerated), there is only one temporal operator (a previous moment operator) and the function symbols of predicate logic are not allowed – only constant symbols are. Furthermore the previous moment operator may not be nested in a formula.

The syntax and semantics of formulae is quite standard. `&` is used for conjunction, `#` for disjunction, `~` for negation, `->` for implication and `<->` for equivalence. The keywords `ALL`, `SOME` and `PRE` are used for universal quantification, existential

quantification and the previous-moment operator, respectively. There also relational operators = and <> for equality and inequality.

As the notation uses past-time modalities, the truth value of a formula at a particular moment of time depends only on the finite number of moments from the initial moment.

A GTO model file is made up of a number of declaration statements. There are statements that

- Declare types (or "sorts").
- Declare types of constants, variables and predicates.
- Declare what predicates represent input or output.
- Define predicates in terms of formulae.
- Define predicates in terms of their true instances.
- Declare invariant formulae (or axioms).
- Include sub-models (files).

An invariant statement takes the form of a single formula. Such formulae are taken as axioms – i.e. to be true at every moment of time. A predicate definition statement has the form $p(x_1, \ldots, x_n)$==*formula*. A definition formula is restricted in that it may not depend on $p$ at the same moment of time. From a logical point of view it can be understood as an axiom stating the equivalence between the defined predicate and its defining formula, but it also has an operational significance for simulation (see section 2.3.2).

The other statements will be explained informally in section 3.

### *2.3 Tools*

The toolset includes the principal formal specification and verification tool GTO, as well as auxiliary tools to translate from other formalisms into the GTO notation and to check models for logical equivalence.

The basic user interface of GTO is a traditional line-oriented one, where the user enters text commands and the response is likewise given as text. There is also a simple graphic user interface. All examples in this paper will use the line-oriented user interface.

The GTO tool has a notion of *state*, which is a truth value assignment to predicates at a particular moment of time (referred to as the "present time") and at the moment preceding the present time. As the previous moment operator may not be nested, it is sufficient to keep the truth value assignment one moment back in time in order to determine the truth values of formulae.

The GTO tool carries out three basic functions:

#### *2.3.1 Static analysis*

After reading a model, GTO performs a static analysis, including type-checking the model and checking that proper declarations and definitions are made.

GTO also finds any defined predicates where the definitions do not depend on any previous moments of time or on input predicates. The truth value assignment

for such predicates are independent of the moment of time and is computed once and for all. The tool also checks that invariants which do not depend on any previous moments of time or input predicates hold.

### *2.3.2 Simulation*

Simulation is a function where the tool incrementally constructs a sequence of states consistent with the model. The simulation feature is made possible by the choice of past-time temporalities rather than the usual future-time ones. When every formula depends only on past moments of time, the tool can determine the truth values of predicate instances at successive moments in time using only information about truth values from the previous moments. That is, a "time line" (or *path*) of truth assignments can be incrementally constructed, moment for moment.

At every simulation step, the state of the tool is updated so that the present-time truth assignment becomes the past-time truth assignment and a new present-time truth assignment is computed according to a *computation rule* for every predicate. Essentially, the tool becomes an interpreter for a synchronous programming language such as Lustre [8].

The computation rules are primarily given by the predicate definitions. The values of instances of defined predicates are computed from the values of their defining formulae. As the definition of a predicate may not depend on the value of the predicate itself at the same moment of time, it is always possible to compute it given the truth values of the previous moment and truth values of other predicates at the present moment.

The computation rule for predicates representing input from the environment is that they keep the same value from the previous moment to the next unless modified by the user in connection with the simulation step.

If some predicate `p` is neither defined nor declared as input, GTO attempts to obtain a computation rule by automatically constructing a definition for the predicate. The definition is constructed from some invariant(s) involving the predicate so that it will always satisfy the invariant(s). If there are invariants

`ALL x1...ALL xn(p(x1,...,xn)<->...)`

(or similar equivalences with `p` to the right), GTO will pick an arbitrary one and turn it into a definition

`p(x1,...,xn)==...`

Otherwise, if there are implications

`ALL x1...ALL xn(p(x1,...,xn)->...)`,

GTO will combine them all into one and strengthen it to an equivalence, e.g. from

`ALL x(p(x)->q(x))` and `ALL x(p(x)->r(x))`,

the definition

`p(x)==q(x)&r(x)`

would be constructed. If none of these cases are applicable, GTO will look for implication invariants with `p` to the right of the implication and process them similarly.

This procedure is called "completion" as it attempts to provide a complete set

of computation rules for the predicates. If some undefined, non-input predicate can not be completed, then the model can not be simulated.

### *2.3.3 Proof*

GTO can carry out an automatic proof of a formula $F$ from the current model, i.e. determine if $F$ is a logical consequence of the set of axioms and definitions. The notion of "logical consequence" is that for every sequence of moments of time – a *path* in the terminology of linear time logic — such that the axioms are true at every moment, $F$ should also be true at every moment.

The proof is done by translating the model and $F$ into propositional logic. As quantification is over finite sets, such a translation is always possible. References to predicates at the previous moment of time are considered to be references to different distinct predicates. The restriction on nesting the previous moment operator ensures that there will be at most one such "previous-moment" version of each predicate. As there is no connection with a particular moment, the effect is an implicit universal quantification over all moments.

The resulting proof problem involving the translated formulae is sent to the separate SAT solver. If the proof fails, the SAT solver produces a counterexample which is read back into GTO and used to set its state. This state can then be examined by the user to find the reason why the proof failed.

However, the proof problem as solved by GTO does not correspond exactly to the notion of logical consequence as defined above. The SAT solver checks one arbitrary, but singular, moment of time. There is no assurance that there is an actual path from the initial moment of time in which the axioms hold at every moment. In other words, a proof may fail because $F$ is false in a state where the axioms are all true, but the state is not on any path where the axioms are true at each previous moment – an *unreachable state*. The consequence is that the proof method is sound but incomplete.

To obtain a complete proof method, the user of GTO must employ a technique which is essentially mathematical induction, using separate base case and induction step proofs. In the base case, $F$ is proved assuming a formula $I$, which characterises the initial state(s) of the system. In the induction step case, $F$ is proved assuming that it holds at the previous moment of time. In this way, unreachable states will be excluded from the analysis.

Just as the case is with ordinary mathematical induction, it can happen that the formula to prove is not strong enough to serve as induction hypothesis in the step case. In that case, a stronger formula must be proved by induction and the requirement formula shown to follow from it by separate proof.

## *2.4 Method*

The steps and information flow of the method is outlined in figure 1. These steps are illustrated in section 3 below.

The specification model is developed from informal requirements or by translation from other formal notations. An important part of the method is the development of a formal theory of the application domain to facilitate developing and

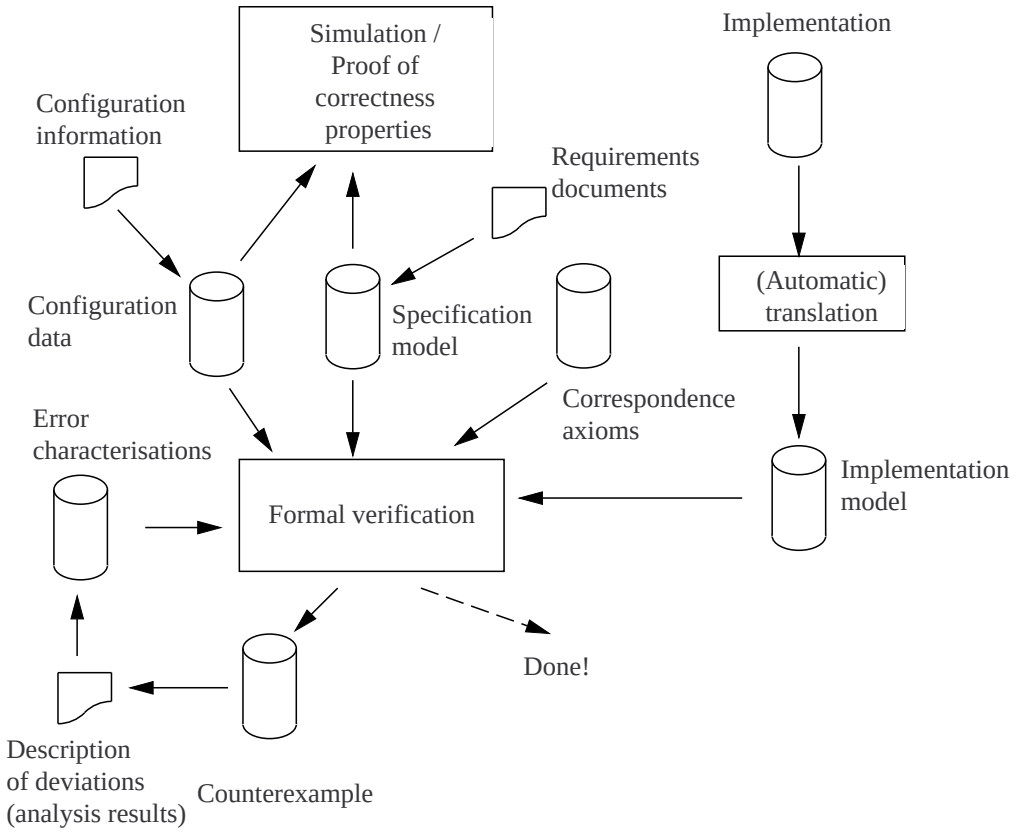


Fig. 1. Information flow in the GTO method

using generic specification models. A description of this is outside the scope of this paper – the interested reader is referred to [6].

As part of the validation of the specification model it can be analysed by the GTO tool by simulation or by proving that it has some desired correctness properties. If the specification is written as a generic specification, configuration data must be added to specialise the specification for a particular system a before it can be analysed. In particular, quantification can only be done over finite sets of objects, so those sets must be defined. Typically the analysis is done for several different configurations. Analysis of the specification in the general case must be done by a different tool which can handle undetermined sets, e.g. Isabelle or PVS [5].

The implementation model is obtained from a description of the implementation by some kind of translation procedure. The GTO toolset includes two auxiliary tools to carry out this translation automatically for two common cases, that of a PLC (programmable logic controller) program written in the STEP 5 language or that of a relay circuit schematic.

The relay circuit translator is important as the major application of the GTO toolset has so far been railway signalling. Relay implementations are still common in this domain – new relay systems are even being commissioned today. The translator takes as input a textual description of the connections in the relay circuit. It

[5] The verification system PVS has been used to prove general correctness properties about GTO specification models.

uses a graph-rewriting technique where individual translation steps are done by rewriting the graph representing the relay circuit. By changing the rewrite rules, the translator can be adapted to different kinds of circuit principles which may require different translation strategies.

There have also been promising attempts to automatically interpret CAD (Computer Aided Design) drawings of relay circuits, eliminating the very tedious and error-prone manual step of creating a textual description.

As the specification is generic, the representation of the state of the environment of the system will generally be different between the specification and implementation models. Before verifying the implementation model, the different representations must be related using *correspondence axioms*.

With the specification and implementation models, configuration data, and correspondence axioms available, formal verification can be attempted.

More often than not, a proof fails because the implementation does not quite follow the specification. Apart from genuine errors in the implementation (or the modelling!), the reason can be that the specification actually restricts behaviour more than necessary. This is not unusual when the formal verification is done separately from the development of the implementation.

When the proof of a requirement fails, the tool will automatically create a "counterexample" state representing a situation where the system exhibits a behaviour not allowed by the specification. By itself the counterexample is of limited use, partly because of its excruciating detail, but also because it does not *explain* the reason for the incorrect behaviour. Most of the information described in the counterexample is coincidental to the particular requirement being violated. E.g. the counterexample will include state information of parts of the implementation not related to the failing function.

An important step of the method is the analysis of a counterexample state to provide a characterisation of the situations where the system exhibits an erroneous behaviour. This analysis is presently carried out by hand and requires a substantial understanding of how the verified system works. The result of this analysis is used as output from the verification process to describe a particular problem with the verified system.

The characterisation is also used to find other situations where the same (or other) requirement is violated. The theorem proving attempt is repeated with the assumption that the recently characterised error situation can not occur. This will cause the theorem prover to disregard that particular situation and it will either report that the proof succeeded (in which case all errors have been found), or it will generate a counterexample for a different error situation which can in its turn be analysed. By successively analysing counterexamples and retrying the proof in this manner, all ways in which the system violates the requirement will eventually be found.

### *2.5 Reliability issues*

The results of formal verification is typically used as evidence of the correctness of the implementation, e.g. in the safety case of a safety-critical system. This raises

the issue of the validity of the evidence itself. How can we rely on the formal verification to produce correct results?

In the case of GTO this issue has been addressed when necessary by a combination of diversity, and of formal verification of the tool itself. The core functionality of the GTO tool has been formally verified using the LPTP [13] verification system for Prolog programs. Diversity is introduced by independently making two models of the implementation – repeating both any manual steps such as hand-coding of relay circuit diagrams and in some cases even duplicating translation tools – and comparing the results. Also, as GTO is not bound to any particular SAT solver, the analysis can be repeated using two different SAT solvers.

# 3 Example

## *3.1 The application*

We will illustrate the method by running through a simple example. The application is a control unit for a crane motor. The control unit has three inputs, *upbutton*, *downbutton* and *stopbutton*, connected to push-buttons. It has two outputs, *moveup* and *movedown* which control the motor. When the "up" or "down" button is pressed, the corresponding output should be activated. The output will continue to be activated until the "stop" button is pressed. The control unit must be designed so that both outputs can not be activated at the same time as that will damage the motor.

## *3.2 Implementation model*

Suppose that the control unit is implemented according to the following pseudo-code program:

**bool** *upbutton,downbutton,stopbutton,moveup,movedown*
*moveup* ← *movedown* ← **false**
**repeat**
    **input** *upbutton,downbutton,stopbutton*
    **if** *downbutton* **then** *movedown* ← **true**
    **if** *moveup* **then** *movedown* ← **false**
    **if** *upbutton* **then** *moveup* ← **true**
    **if** *movedown* **then** *moveup* ← **false**
    **if** *stopbutton* **then** *moveup* ← *movedown* ← **false**
    **output** *moveup,movedown*
**end repeat**

A sample GTO model of the implementation above is

```
PRED upbutton, downbutton, stopbutton, moveup, movedown, start;
INPUT upbutton, downbutton, stopbutton;
OUTPUT moveup, movedown;
movedown == (downbutton# PRE movedown)&~stopbutton& PRE ~moveup;
moveup == (upbutton# PRE moveup)&~stopbutton&~movedown;
start == ~movedown & ~moveup;
```

The first line of the model declares the predicates used. These predicates lack arguments, i.e. they are propositional variables. The next two lines declare the predicates used to represent input and output. The predicates `moveup` and `movedown` are defined by formulae representing the effects of executing the loop body of the pseudo-code program. The `start` predicate is used to characterise the initial state of the system before the first iteration of the loop, after both `moveup` and `movedown` have been set to false.

This model can be simulated by GTO to investigate its behaviour. Text following the `>` symbol is input to GTO, other text is output from GTO.

```
> init
> do upbutton
moveup
> do ~upbutton
> do downbutton
> do ~downbutton
> do stopbutton
~moveup
> do ~stopbutton
> do downbutton
movedown
```

The simulation is initialised by the `init` command which sets the initial truth values of all predicates to false. New truth values are then computed according to the definitions. With the `do` command, the user asks for one time step to take place with the input predicate `upbutton` set to true. This causes the output predicate `moveup` to be true. The user then resets `moveup` to false ("releasing" the up button) in the next step - this does not cause any changes to the output predicate values.

The user simulates a press and release of the down button. This does not have any effect as the up output is already activated. Pressing the stop button causes `moveup` to become false. Now pressing the down button will make `movedown` true.

### *3.3 Specification*

We will make a formal specification of two safety properties of the control unit – that both outputs must not be active at the same time, and that between the times the cancel button has been pressed and the next time a request button is pressed, the corresponding output must not be active.

To illustrate generic specifications, we will make the requirements specification more general. The specification will express the safety properties for a control unit which can handle any number of functions, not just the "up" and "down" functions of the sample implementation. Also, the specification can be tailored to exclude any particular pair of outputs from being active at the same time while other outputs are independent.

```
TYPES function;
VAR f,f1:function;
PRED request(function), activate(function), cancel,
```

```
   cancelrequest(function), exclude(function,function), initial;
INPUT request,cancel;
OUTPUT activate;
ALL f (activate(f) -> ALL f1 (exclude(f,f1) -> ~activate(f1)));
cancelrequest(f) == cancel # PRE cancelrequest(f) & ~request(f);
ALL f (activate(f) -> ~cancelrequest(f));
ALL f ALL f1 (exclude(f,f1) -> exclude(f1,f));
initial == ALL f cancelrequest(f) & ~cancel & ALL f ~request(f);
```

The specification begins with a declaration of the type `function` representing the set of different functions that the specified system controls. Next, the variables `f` and `f1` are declared to range over objects of the type `function`. On the following lines, various predicates and their argument types are declared, e.g. `request` is a predicate with one argument of type `function`.

The first invariant states that if a function is activated, then any other function which is excluded from being active at the same time must, in fact, be deactivated. `cancelrequest` is defined to be true for each function from the time the `cancel` input is true until it becomes false, and the corresponding request input becomes true. The second invariant states that an output must not be activated during the time interval that `cancelrequest` is true for that function. The third invariant is a consistency check of the `exclude` predicate, stating that it must be a symmetric relation.

Note that the `exclude` predicate is not defined, nor are constants of the type `function` given. This information is provided as configuration data when the specification is used with a particular implementation. E.g. in this case:

```
CONST up,down:function;
FACTS exclude(up,down),exclude(down,up);
```

The specification is configured for use with the crane motor application by declaring the constant symbols `up` and `down` to belong to the type `function` and defining the predicate `excludes` to be true in exactly the stated instances. When loading the specification into GTO, the symmetry of the `exclude` predicate according to the third invariant above is checked and if there is a discrepancy, it is reported.

We now analyse the specification model by simulation. The example specification is nondeterministic as it only describes safety properties which can be satisfied in different ways. There are no requirements on when outputs should be active, only when they should not be. In this case, GTO can "complete" the model by constructing a definition for `activate`, making the model deterministic. The constructed definition is

```
activate(f) == ALL f1 (exclude(f,f1)-> ~activate(f1)) &
               ~cancelrequest(f);
```

This definition cases outputs will be activated when it is "maximally dangerous" (and presumably most interesting) with respect to the specification.

We want to start the simulation in a particular initial state where we assume that no request inputs are active, but that the cancel input has been active. This

state is characterised by the `initial` predicate.

```
> satisfy initial
The formula is satisfiable.
> do request(up)
activate(up)
> do ~request(up)
> do request(down)
> do ~request(down)
> do cancel
~activate(up)
> do ~cancel
> do request(down)
activate(down)
```

The interaction begins with the user requesting GTO to find a state in which `initial` is true, i.e. the specification is in the initial state. The SAT solver is called upon to find such a state. The interaction then goes on along the same lines as the simulation of the implementation model. Note that the completed definition of the `activate` predicate makes GTO automatically activate the appropriate output when allowed by the specification.

### *3.4 Formal verification*

Before verification, a composite model must be created including both implementation and specification as well as the correspondence axioms:

```
USE implementation;
REFINES specification;
ALL f (request(f) <-> f=up&up_button # f=down&down_button);
ALL f (activate(f) <-> f=up&moveup # f=down&movedown);
cancel <-> stop_button;
```

Assuming that `implementation` and `specification` are names of files containing the implementation and specification models, respectively, the first two lines state that this composite model includes the implementation and specification with the purpose of refining the specification model. This has the important effect of *not* taking invariants in the specification files as axioms, as they are the requirement formulae to be proved.

After loading the composite model, proofs of the requirement formulae can be done. These formulae are identified by the name of the file where they occur, and a sequence number. The following interaction shows that the first requirement formula of the specification is satisfied by the implementation.

```
> prove specification_1
The formula is valid
```

The second requirement formula can not be directly proved. It is true only when the system is initialised in a state where the requirement already holds. Instead, it has to be proved by the induction proof method described in section 2.3.3. As the

system is in its initial state when the predicate `start` is true, the induction proof can be done as:

```
> prove start -> specification_2
The formula is valid
> prove PRE specification_2 -> specification_2
The formula is valid
```

Now, suppose that a mistake had been made in the implementation, e.g. that the pseudo-code statement **if** *movedown* **then** *moveup* ← **false** had been omitted. In that case the definition of `moveup` would become

```
moveup == (upbutton# PRE moveup)&~stopbutton;
```

and the proof attempt would fail

```
> prove specification_1
The formula is falsifiable.
> why
Formula is FALSE because
f=up, activate(up)=>... f1=down, exclude(up,down)=>... activate(down)
> evf movedown & upbutton
TRUE
```

As the formula can not be proved, GTO creates a state in which it is false. Using the `why` command, the user asks GTO to attempt an explanation of why the requirement could not be proved. The tool gives instances (witnesses) for the quantified variables for which the quantified formula is false and displays the corresponding instance of the quantified formula. In this case with only two objects of the type `function` it is obvious that each of them have to be involved, but in more complicated cases the explanation function can be of great help. The user can also inspect the state directly e.g. by using the `evf` command to evaluate a formulae in it.

After inspecting the state and implementation, the user identifies the error — that if the crane motor is already moving down, then pushing the up button will activate the up output. The error situation can be described by the formula `movedown&upbutton`. To find other possible problems with the implementation, the proof is repeated with the error situation excluded by assuming the negation of its description:

```
> prove ~(movedown&upbutton) -> specification_1
The formula is valid.
```

The proof succeeds, so that was the only error with respect to this requirement.

## 4 Conclusions and future work

We have outlined the function of the GTO toolset and how it is used for formal specification and verification work. The goal of automating the time-consuming parts of the process has to a large extent been realised.

The major task which remains manual is the analysis of counter-examples from failed proof attempts. Although it is unlikely that this process can be completely

automated, the tool could provide substantial support to the user in finding explanations of why a requirement fails and in characterising the situation in which this happens.

# References


[1] Benveniste, A. and Berry, G., The Synchronous Approach to Reactive and Real-Time Systems, Proceedings of the IEEE, Vol. 79 No. 9, pp. 1270–1282 (1991).

[2] Biere, A., Limmat, http://fmv.jku.at/limmat.

[3] Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M., NuSMV: A new symbolic model checker, International Journal on Software Tools for Technology Transfer, Vol. 2 No. 4, pp 410–425 (2000).

[4] Clarke, M.E., Grumberg, O. and Peled, D.A., Model Checking, MIT Press (1999).

[5] Eriksson, L-H., Using Formal Methods in a Retrospective Safety Case, In Heisel, M., Liggesmeyer, P., Wittmann, S. (eds.): Computer Safety, Reliability, and Security – 23rd International Conference, SAFECOMP 2004, Springer Lecture Notes in Computer Science 3219, Springer-Verlag (2004).

[6] Eriksson, L-H., Use of Domain Theories in Applied Formal Methods, Technical Report 2006-029, Uppsala University, Dept. of Information Technology (2006).

[7] Groote, J.F. and Warners, J.P., The Propositional Formula Checker HeerHugo, Technical Report SEN-R9905, Centre for Mathematics and Computer Science (CWI), Amsterdam (1999).

[8] Halbwachs, N., Caspi, P., Raymond, P. and Pilaud, D., The synchronous dataflow programming language Lustre, Proceedings of the IEEE, Vol. 79 No. 9, pp. 1305–1320 (1991).

[9] Huber, M. and King, S., Towards an Integrated Model Checker for Railway Signalling Data, In Eriksson, L-H. and Lindsay, P.A. (eds.): FME'2002: Formal Methods – Getting IT Right, pp. 204–223, Lecture Notes in Computer Science 2391, Springer-Verlag (2002).

[10] Jackson, D., Alloy: A Lightweight Object Modelling Notation, ACM Transactions on Software Engineering and Methodology, Vol. 11 No. 2, pp. 256–290 (2002).

[11] Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L. and Malik, S., Chaff: Engineering an Efficient SAT Solver, Proceedings of the 38th ACM/IEEE Design Automation Conference (DAC'01), pp. 530–535, ACM/IEEE (2001)

[12] Sheeran, M. and Stålmarck, G., A Tutorial on Stålmarck's Proof Procedure for Propositional Logic, In Gopalakrishnan, G. and Windley, P. (eds.): Proc. 2nd Intl. Conf. on Formal Methods in Computer-Aided Design, FMCAD'98, pp. 82–99, Lecture Notes in Computer Science 1522, Springer-Verlag (1998)

[13] Stärk, R.F., The Theoretical Foundations of LPTP (a Logic Program Theorem Prover), The Journal of Logic Programming, vol. 36 pp. 241-269 (1998).

[14] Zhang, H., SATO: An Efficient Propositional Prover, In McCune (ed.): Proc. 14th International Conference on Automated Deduction (CADE-14), Lecture Notes in Computer Science, Springer-Verlag (1997)