ClawZ: Cost-Effective Formal Verification for Control Systems

M.M. Adams and P.B. Clayton

Systems Assurance Group, QinetiQ
m.adams@eris.qinetiq.com, p.clayton@eris.qinetiq.com

Abstract. Control system software now plays a key role on many platforms, including aircraft and automobiles. However, as control system software has been performing increasingly complex tasks, the associated software development, maintenance and certification costs have escalated significantly. The ClawZ toolset is dedicated to the formal verification of control system software. By using some novel ideas, it achieves the highest levels of assurance whilst not suffering from the prohibitively high costs normally associated with applying formal verification. It has been successfully used in the certification of the Flight Control Computer of the Eurofighter Typhoon aircraft. This paper outlines the toolset, and explains how the approach used to build it enables formal verification costs to be dramatically reduced whilst not compromising on soundness.

Keywords: industrial formal verification, refinement, formal proof, Z, ProofPower, safety-critical software, real-time software, control systems, Simulink, Ada, Eurofighter Typhoon.

1 Introduction

1.1 Control Systems

A *control system* is a mechanism for controlling physical attributes of a platform, according to user inputs and measured physical attributes of the platform and its environment. Control systems occur in a wide range of platform domains, from higher-end domains such as aircraft, automobiles and nuclear and chemical plant, down to simpler lower-end domestic domains such as central heating systems and washing machines.

Over the past 30 years, digital control systems have replaced analogue electronic control systems in virtually all domains. This has given considerable scope for making control systems more sophisticated, and control systems in higher-end domains have indeed been getting increasingly complex. For example, Flight Control Computers (FCCs), that control an aircraft's flight surfaces to give it stability and manoeuvrability in the air, have become increasingly sophisticated in fighter aircraft, to achieve improved manoeuvrability and a reduction in the pilot's workload.

However this increased complexity comes at a cost. Many of these higher-end domains, such as aircraft and nuclear plant, are safety-critical and thus platforms have to be developed to high standards and then certified to be safe. The costs of developing, maintaining and certifying control system software for these platforms have escalated so much that together they can now represent a large proportion of overall platform costs. Other domains, such as automobiles, involve high-volume products, and reliability problems with safety-related features are requiring expensive product recalls, that are cutting deep into manufacturers' low profit margins.

1.2 Simulink

The use of graphical control system design tools is helping to improve the situation. Instead of designing control systems on paper and then only finding problems further down the line during software testing, control system design engineers are able to simulate the behaviour of the control system at the design stage, finding problems with the functionality and performance. This saves on expensive iterations of the software development cycle. The most widely used graphical control system design tool is Simulink [1].

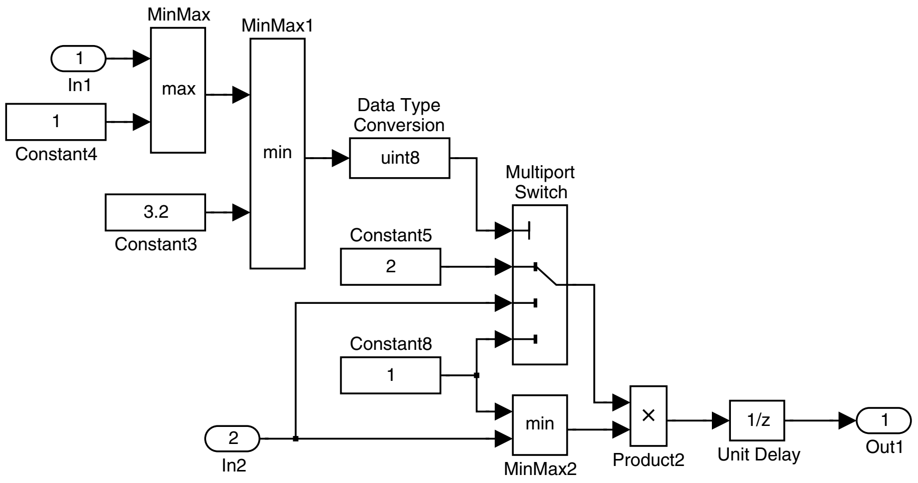


Fig. 1. The Simulink diagram for Pump

In Simulink, a control system specification is expressed in a graphical form called a *Simulink diagram.* A Simulink diagram consists of a collection of inputs, outputs and blocks, all connected by wires. Each block also has its own inputs and outputs, and is either a basic block, corresponding to a primitive operation such as sum or multiplication, or a subsystem block, corresponding to a subdiagram. Opening a subsystem block will reveal the sub-diagram and its internal blocks and wires. See Figure 1 for a simple example of a Simulink diagram.

1.3 Formal Verification

Formal verification often plays an important role in the certification of safety-critical platforms. This is used because it provides very high levels of assurance for finding certain important classes of software error. However, the tools required for performing formal verification are normally prohibitively expensive to use. This is due to two main factors: they require highly-skilled expert analysts, and they require a high degree of user interaction. The costs of using formal verification tools are so high that formal verification, despite the useful extra assurance it can offer, is rarely performed outside safety-critical domains, and, even within safety-critical domains, mainly only for the certification of software.

ClawZ is a toolset for the formal verification of software[1]. It differs from other formal verification toolsets in that it is dedicated to a specific verification domain. In particular, it is dedicated to the verification of control system software that has been specified in Simulink and implemented in Ada. By taking advantage of this fact, the toolset enables analysts to interact with it at a more abstract level than with most formal verification toolsets. This enables the analyst's expertise to be reduced, whilst at the same time considerably reducing the amount of user interaction required. This can result in dramatically reduced formal verification costs. Despite the cost savings, the assurance gained by using ClawZ has in no way been compromised, and in fact it achieves even higher assurance than many existing formal verification toolsets. In principle, the approach used to build the toolset is equally applicable to other verification domains.

ClawZ has been successfully used in the certifications of three successive versions of Eurofighter Typhoon's FCC, one of the most advanced FCCs yet produced. Its core control system functionality is implemented in 35,000 non-comment, non-blank lines of Ada source code. Analysts were typically able to formally verify around 50 to 80 lines of source code per day each. Although an accurate comparison is difficult due to the differing nature of individual verifications, this clearly compares favourably with a typical 10 to 15 lines of source code per day for using other toolsets in other industrial formal verifications (for example, see [2, Overview, page 16]).

1.4 Overview

This paper outlines the ClawZ toolset, and explains how the approach used to build the toolset enables formal verification costs to be dramatically reduced, whilst not compromising on soundness. Rather than by appealing to a theoretical framework, this claim about cost savings is backed up by pointing to industrial, albeit anecdotal, experience of using the toolset.

In Section 2, an overview is given of how a ClawZ analysis is performed from the perspective of the analyst. In Section 3, the architecture of the ClawZ toolset is explained, as well as how this architecture ensures a high level of assurance. Conclusions are drawn in Section 4.

[1] The name ClawZ was previously used to refer to one component of the toolset, which is now called Z Producer.

```
procedure PUMP (COND  : in     REAL_T;
                STATE : in out REAL_T;
                GO    :    out REAL_T)
is
   TMP : REAL_T;
begin
   GO := STATE;
   if COND > 1.0 then
      TMP := 1.0;
   else
      TMP := COND;
   end if;
   if PUMPS_GO < 2.0 then
      STATE := 2.0 * TMP;
   elsif PUMPS_GO < 3.0 then
      STATE := COND * TMP;
   else
      STATE := TMP;
   end if;
end PUMP;
```

Fig. 2. The Ada subroutine implementation for Pump

2 Using ClawZ

In this section, an overview is given of how a ClawZ analysis is performed from the perspective of the analyst. The intention is to give an impression of the analyst tasks involved in using the toolset, and of the required degree of analyst interaction and expertise. This should give the reader an understanding of how formal verification costs are so significantly reduced when compared with traditional formal verification toolsets.

Throughout this section, statistics and screen shots are used to help give a feel for the degree of analyst interaction required in a typical analysis. The quoted statistics are for the analysis of a typical subroutine of 80 non-comment, non-blank lines of Ada source code. Any references to computer execution time relate to using a 2.2GHz Pentium IV processor with 1GB of RAM. The statistics are rough figures, intended as a guide only, and are based on experiences from the industrial analyses of Typhoon's FCC as well as numerous other smaller analyses. The true figures will vary, depending on the peculiarities of the system being analysed. The screen shots are from the analysis of a non-trivial implementation of the Simulink diagram in Figure 1. See Figure 2 for this implementation, written in 20 lines of Ada. Note that the reader is not expected to understand the detail of these screen shots.

A ClawZ analysis verifies that a Simulink specification of a control system is correctly implemented in Ada, and involves performing a separate analysis for each Ada subroutine that implements part of the Simulink diagram (called a *control system subroutine*). Note that the Ada source code may contain some subroutines that only perform tasks outside the scope of what is specified in the Simulink diagram. Such subroutines are not covered by a ClawZ analysis. Also note that a ClawZ analysis just covers core functional correctness of the source code, and does not cover classes of error such as run-time errors (e.g. overflow or divide-by-zero) or program termination.

The analysis of a given subroutine breaks down into three principal stages: specification, witnessing and interactive proof. The analyst skills required differ between the stages, and analysts should be assigned to stages according to their skill sets. The underlying mathematical model used in ClawZ is expressed in the Z notation [3], and all analysts are required to be familiar with reading this notation.

The graphical user interface to the ProofPower theorem prover [4], called `xpp`, is used throughout an analysis, for viewing and editing Z, and for giving interactive feedback during the witnessing and interactive proof stages. It also helps to use Simulink, in conjunction with the toolset, to view Simulink diagrams, although strictly speaking this is not necessary if a printout can be obtained.

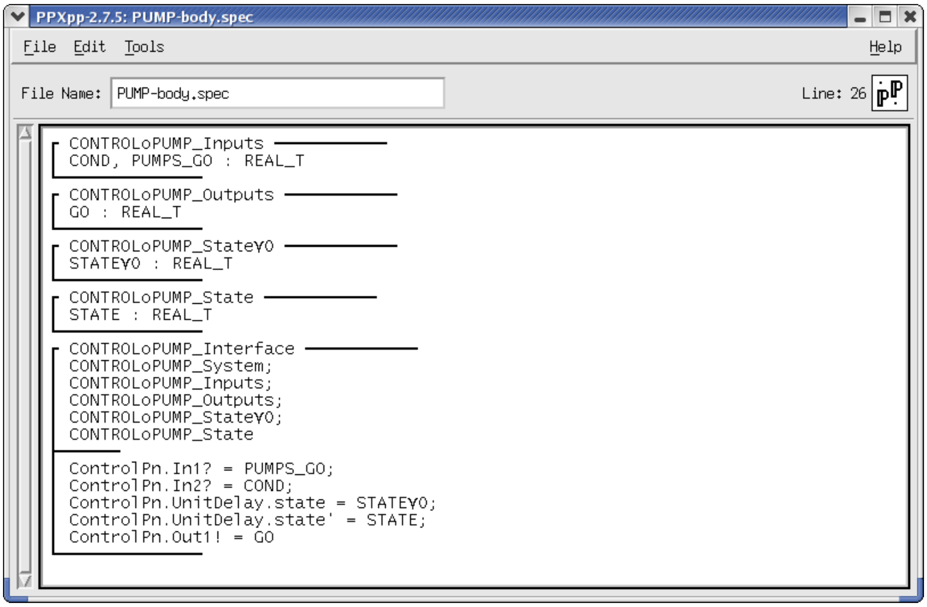
2.1 The Specification Stage

In the specification stage, the Ada subroutine's specification components are created. These capture the intended behaviour of the subroutine in terms of the Simulink diagram and the input and output parameters of the Ada subroutine.

Firstly, the analyst produces the subroutine's *block list*, that lists exactly those blocks in the Simulink diagram that are intended to be implemented by the Ada subroutine. An Ada subroutine will often correspond to a single Simulink subsystem block, in which case it is sufficient for the analyst simply to identify just this one Simulink block, although sometimes it is necessary to identify numerous blocks. The collection of the parts of the Simulink diagram that correspond to the block list is called an *artificial subsystem block*, since it can be considered to have its own inputs, outputs and internal blocks although it does not itself necessarily correspond to an actual Simulink subsystem block. The process of identifying the subroutine's blocks can be helped by examining the software's design documentation.

Secondly, the analyst defines a data refinement retrieve relation between all of the artificial subsystem's inputs and outputs, and corresponding input and output parameters of the Ada subroutine. This will often be a simple one-to-one mapping. See Figure 3 for an example data refinement relation. The left-hand sides of the equalities in the data refinement relation refer to Simulink wires, and the right-hand sides refer to Ada variables.

Finally, the analyst may have to identify a precondition for the Ada subroutine. Usually a subroutine will assume nothing about the variables it reads. However, some subroutines make assumptions, and these need to be explicitly identified as preconditions in order to complete the subroutine's verification. These usually only become apparent during the witnessing stage, and need to be justified once identified. Often a subroutine's precondition will be established



```
CONTROLoPUMP_Inputs
COND, PUMPS_GO : REAL_T

CONTROLoPUMP_Outputs
GO : REAL_T

CONTROLoPUMP_StateY0
STATEY0 : REAL_T

CONTROLoPUMP_State
STATE : REAL_T

CONTROLoPUMP_Interface
CONTROLoPUMP_System;
CONTROLoPUMP_Inputs;
CONTROLoPUMP_Outputs;
CONTROLoPUMP_StateY0;
CONTROLoPUMP_State

ControlPn.In1? = PUMPS_GO;
ControlPn.In2? = COND;
ControlPn.UnitDelay.state = STATEY0;
ControlPn.UnitDelay.state' = STATE;
ControlPn.Out1! = GO
```

Fig. 3. *CONTROLoPUMP_Interface*, the data refinement relation for Pump

in another control system subroutine. Such a precondition will then become a postcondition of the subroutine that establishes it.

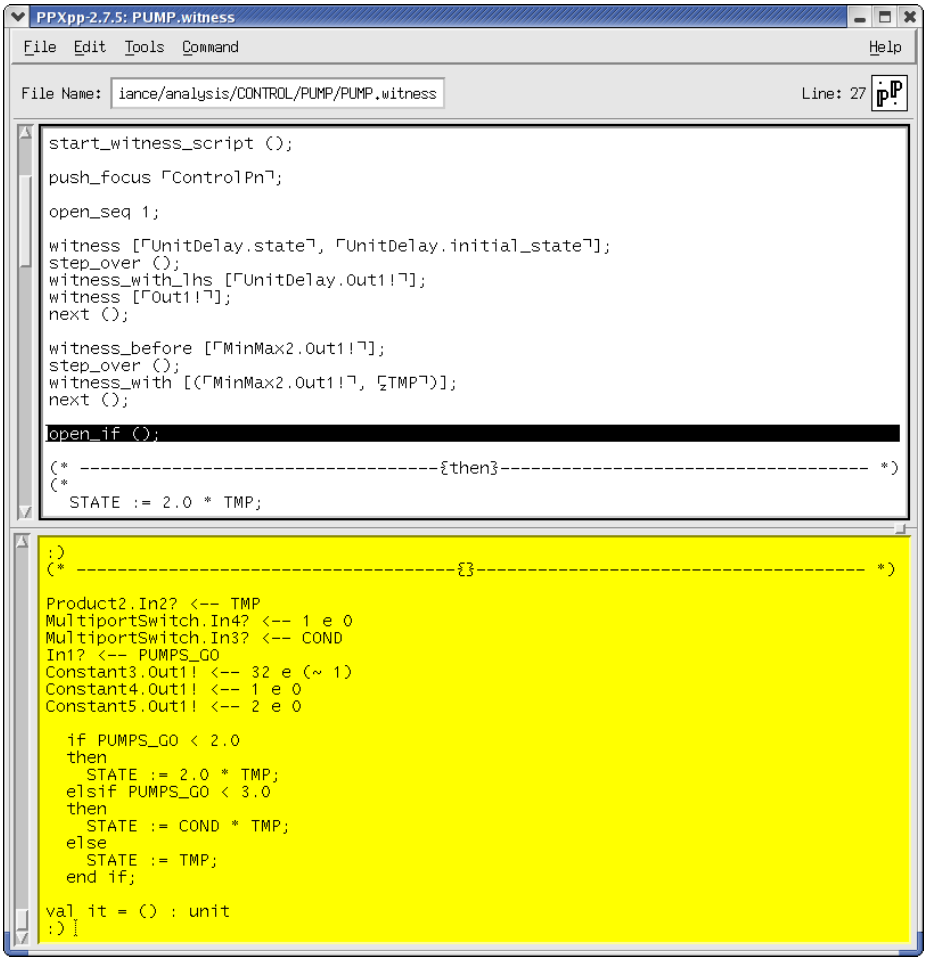
The specification stage will typically require a total of around 30 to 70 lines of input from the analyst. This will typically take around half an hour to an hour. The analyst performing the specification stage needs to be able to read Simulink diagrams and to be aware of the subtleties of writing a formal specification.

The important point to note about this stage is that there is no need for the analyst to laboriously construct a detailed specification of an Ada subroutine's behaviour. The analyst just provides the key components of the specification, and the toolset will fill out the details automatically (see Section 3.1).

2.2 The Witnessing Stage

The witnessing stage takes up the bulk of the analysis effort. In this stage, the analyst constructs a *witness script*, which justifies in detail how the Ada subroutine body correctly implements its specification. This is done by identifying correspondences between wires in the Simulink diagram and the values of Ada variables at specific points in the subroutine's body.

The analyst constructs the witness script by interacting with a tool called RSG, via the `xpp` interface. See Figure 4 for an example of an interactive RSG session. The witness script is edited in the upper window, and is entered for interactive RSG feedback, which appears in the lower window.


PPXpp-2.7.5: PUMP.witness
File Edit Tools Command Help
File Name: iance/analysis/CONTROL/PUMP/PUMP.witness
Line: 27

```
start_witness_script ();

push_focus ⌜ControlPn⌝;

open_seq 1;

witness [⌜UnitDelay.state⌝, ⌜UnitDelay.initial_state⌝];
step_over ();
witness_with_lhs [⌜UnitDelay.Out1!⌝];
witness [⌜Out1!⌝];
next ();

witness_before [⌜MinMax2.Out1!⌝];
step_over ();
witness_with [(⌜MinMax2.Out1!⌝, ⓩTMP⌝)];
next ();

open_if ();

(* ----------------------------------{then}------------------------------------ *)
(*
  STATE := 2.0 * TMP;
```

```
:)
(* ------------------------------------{}------------------------------------- *)

Product2.In2? <-- TMP
MultiportSwitch.In4? <-- 1 e 0
MultiportSwitch.In3? <-- COND
In1? <-- PUMPS_GO
Constant3.Out1! <-- 32 e (~ 1)
Constant4.Out1! <-- 1 e 0
Constant5.Out1! <-- 2 e 0

  if PUMPS_GO < 2.0
  then
    STATE := 2.0 * TMP;
  elsif PUMPS_GO < 3.0
  then
    STATE := COND * TMP;
  else
    STATE := TMP;
  end if;

val it = () : unit
:)
```

Fig. 4. The interactive session from the witnessing stage for Pump. The *open_if* command is about to be entered, for witnessing the second Ada *if* statement in Figure 2.

A witness script needs to account for every Simulink wire in the subroutine's artificial subsystem in terms of values of Ada variables, called *witnesses*. RSG ensures that this is done by maintaining a wavefront of wires in the Simulink diagram, where wires behind the wavefront have already been accounted for. The user is prompted to supply a witness for any wire on the current wavefront. RSG enables the analyst to step through the subroutine body to reach an appropriate point for supplying a given witness. When a wire's witness has been supplied, the wavefront advances along that wire. A witness script is complete when the wavefront has crossed the entire diagram, from inputs to outputs.

Often, a suitable witness is self-evident because the Ada source code closely mirrors the detail of the Simulink diagram, in which case RSG will supply an appropriate suggested witness, derived by symbolic execution of the specfication. However, there will usually be a few witnesses that the analyst has to construct manually. This will be when the Ada source code has departed from the detail of the Simulink diagram, perhaps in order to achieve greater efficiency.

A witness script for a typical 80 line subroutine will be around 100 to 150 lines long. This will typically take around two to three hours to construct. However, note that the witnessing stage is not necessarily complete once a witness script has first been completed.

Once a witness script has been completed, the analyst then submits it to the toolset for processing and waits for the results. During this time, the toolset is generating a set of Z conjectures to be proved, called *verification conditions*, and attempting to prove them automatically. If all these verification conditions are provable, then the Ada subroutine meets its specification. The proof side of the processing is carried out by a tool called Supertac. After a wait of typically around a minute or two, the toolset reports back the results in the form of the total number of verification conditions generated and the total number that have been proved automatically. There tends to be almost one verification condition generated per line of source code analysed, and so a typical 80 line subroutine will have around 50 to 70 verification conditions.

If all have been proved automatically, then the analysis for that subroutine is complete and the subroutine has been shown to correctly implement its specification. If there are any unproved verification conditions, the analyst must examine the proof output file. In this file, any unproved verification conditions have been simplified. Supertac ensures that a simplified verification condition will be provable if and only if its original unsimplified verification condition is provable. A simplified verification condition will typically be 5 to 15 lines of Z.

On examining the proof output file, the analyst must grade each simplified verification condition. If every verification condition is graded as provable, then the subroutine is passed on to the interactive proof stage of the analysis. Otherwise, the analyst must determine why there are unprovable verification conditions. This may be due to an analyst error, either in constructing the subroutine's specification components or its witness script. If this is the case, then the analysis is iterated from the relevant stage. However, if a verification condition is unprovable, but this is not due to an analyst error, then a genuine error has been found, either in the Simulink diagram or its Ada implementation.

In practice, most unproven verification conditions will be due to an error in the witness script. The simplified verification condition will contain information about the part of the witness script it relates to, to help the analyst locate the error. The next most common explanation for an unproven verification condition is that it is provable, but Supertac has not managed to prove it completely. Supertac will complete the proof of around 95% to 98% of provable verification conditions. A typical 80 line subroutine will have all but one or two of its provable verification conditions completely proved by Supertac.

There will typically be two to four iterations before all the witnessing errors are ironed out, typically taking half an hour each. The most complex subroutines to analyse will be those that deviate from the structure of the Simulink diagram in many ways, and it will usually take many more iterations to reach a correct witness script for these.

To perform the witnessing stage effectively, the analyst needs to be able to read Simulink diagrams and Ada programs, and to have a good knowledge of how to interact with RSG. These are all skills that are quickly picked up by university software engineering graduates. However, the analyst also needs to be able to assess the provability of Z conjectures, which is a less common skill.

There are two important points to note about this stage. Firstly, due to the relatively abstract level of interaction with RSG, it involves considerably less effort than with the corresponding activity in other formal verification toolsets. The analyst does not have to construct a detailed refinement script, or be familiar with all the subtleties of algorithmic program refinement. Secondly, due to Supertac's high level of proof automation, there can be an easy separation of duties between the witnessing analyst and the interactive proof analyst. The analyst performing the witnessing stage does not have to be familiar with the highly-skilled and time-consuming task of interactive formal proof, and can usually produce a correct witness script, albeit after a few iterations, before any remaining unproved verification conditions are passed on for interactive proof.

2.3 The Interactive Proof Stage

The interactive proof stage involves using the `xpp` interface to construct an interactive proof script for every remaining unproved verification condition for the subroutine. These verification conditions will already have been graded as provable during the witnessing stage.

The verification conditions are proved using the ProofPower theorem prover. Interactive proofs are carried out within a proof environment that closely corresponds to HOL's subgoal package [5]. A proof involves the analyst applying a series of proof tactics to a proof goal. The analyst can make the proof branch into several parts, called *subgoals*. A proof is complete when all subgoals have been reduced to true. As well as having access to ProofPower's tactics for reasoning about Z, the analyst also has access to specialised tactics used in the implementation of Supertac. Occasionally, the analyst performing the interactive proof will notice that a verification condition has been wrongly graded as provable, in which case it will be passed back to the witnessing stage for regrading.

This stage is the most skilled stage to perform, and a thorough knowledge of ProofPower's proof tactics is required. It typically takes 15 to 30 minutes to prove each simplified verification condition, although some can take several hours. However, because 95% to 98% of the verification conditions have been automatically proved, and those that have not have been largely simplified, interactive proof takes up the smallest proportion of the overall analysis effort.

The important point to note about this stage is that, due to Supertac's high level of proof automation, very little of the highly-skilled activity of performing interactive proof is necessary. The amount of interactive proof required is less than a tenth of what is typically required in other formal verificaiton toolsets.

3 The ClawZ Approach

In this section, the components of the ClawZ toolset and how they fit together are described in more detail. As well as explaining how the toolset works, the intention is to give an understanding of how the approach used to build the toolset can be applied to other verification domains.

3.1 The ClawZ Architecture

The ClawZ toolset is composed of six tools: Z Producer [6,4], RSG, DAZ[2] [7,4], Supertac, ProofPower [4] and CPS. The central tool in the toolset is DAZ. This is a tool for performing formal refinement between a Z specification and Ada source code. All the other tools revolve around DAZ. Z Producer translates a Simulink diagram into a Z specification, ultimately used by DAZ. RSG is used as an interactive tool for creating a witness script, but its main role is to translate the witness script into a DAZ refinement script. The Supertac and ProofPower theorem provers are used to prove verification conditions that are output from DAZ as a result of processing a refinement script. Finally, CPS is ClawZ's Unix command line environment, and acts as a front end for much of the analyst interaction with the toolset, as well as performing house keeping tasks. See Figure 5 for a diagram illustrating the architecture of the toolset.

The refinement supported by DAZ is loosely based on Carroll Morgan's algorithmic refinement calculus [8]. A refinement script starts with a *specification statement* for a block of source code. The specification statement defines what the source code block is supposed to do. It consists of three parts: a frame, that lists the variables that are allowed to change in the source code block; a precondition, that can be assumed about the values of variables on entry to the block; and a postcondition, that must be upheld on exiting the block. The pre- and postconditions are expressed in Z.

The specification statement can then be refined to source code under a series of refinement steps. In intermediate refinement steps there will be a mixture of source code and specification statements. Each refinement step results in verification conditions being generated by DAZ. DAZ can output the set of verification conditions and the Ada source code resulting from the refinement. Every verification condition of every refinement step must be proved in order to establish that the source code meets its specification.

Before a subroutine can undergo interactive witnessing with RSG, it requires a specification statement. This is constructed using the components supplied by the analyst during the specification stage (see Section 2.1). The subroutine's block list is read in by CPS, which uses it to construct an input command for Z Producer. Z Producer then outputs a Z translation of the artificial subsystem corresponding to these blocks. CPS then combines this Z artificial subsystem with the subroutine's data refinement relation and any preconditions and post-

[2] DAZ has also been known as the Compliance Tool and the Compliance Notation Tool.

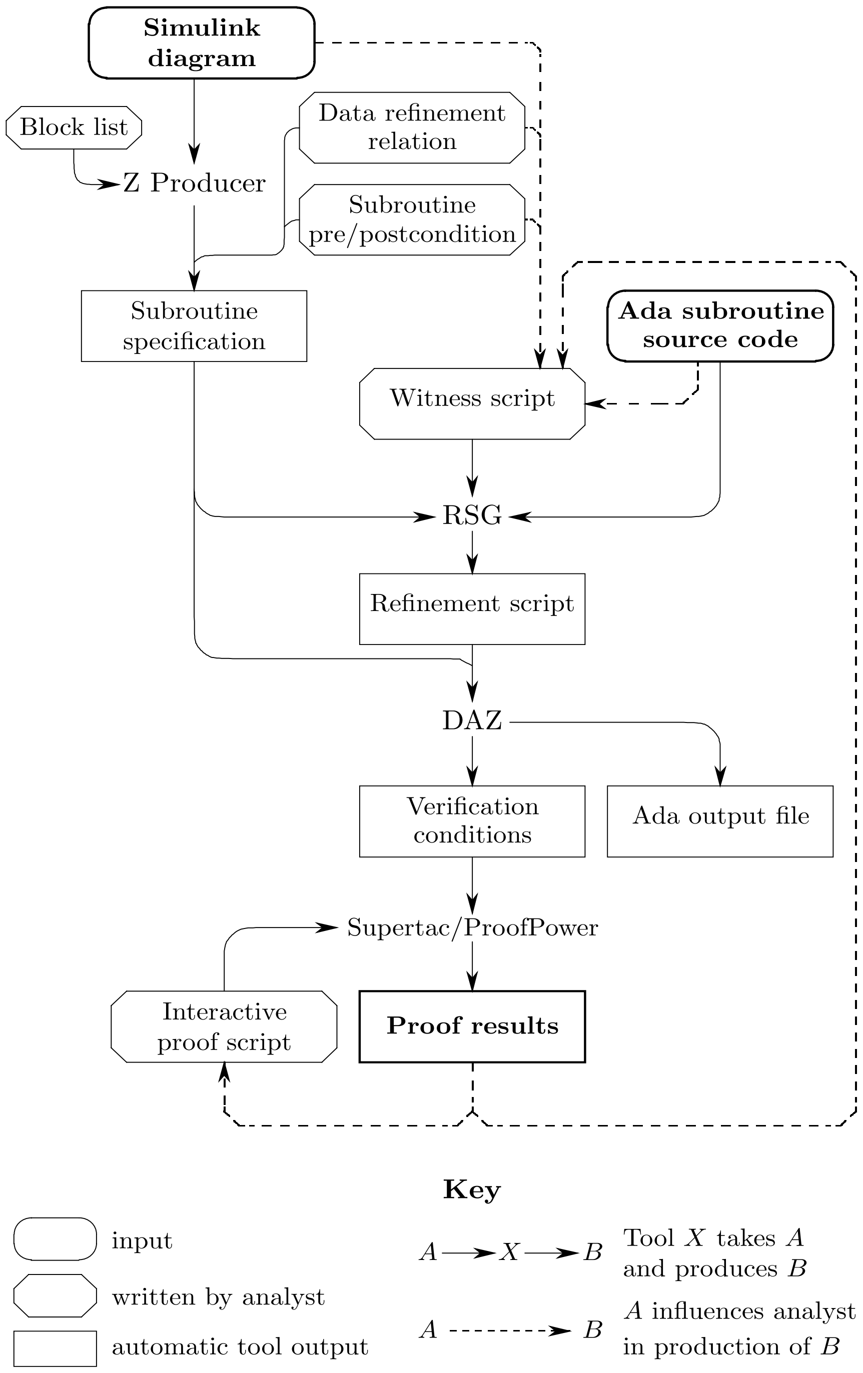


Fig. 5. The architecture of the ClawZ toolset

conditions associated with the subroutine to form a specification statement for the subroutine. This is all done in the execution of one CPS command.

Once a witness script has been produced by the analyst (see Section 2.2), CPS passes it, the subroutine's specification statement and the subroutine's Ada source code to RSG, which results in a DAZ refinement script being generated. For reasons explained in Section 3.2, this refinement script outputted by RSG has the starting point, i.e. the subroutine's specification statement, missing. CPS then combines the subroutine's specification statement with RSG's output to form a complete refinement script. This complete refinement script is then submitted to DAZ, which results in a set of verification conditions and an Ada output file being generated. For reasons explained in Section 3.2, CPS then checks that the Ada subroutine output from DAZ is syntactically equivalent to the original Ada subroutine source code. Supertac is then run on every verification condition generated. Finally, CPS summarises the results of the Supertac run to the screen. Again, this is all done in the execution of one CPS command.

Before the ClawZ toolset was invented, a few parts of control systems were analysed by manually writing DAZ refinement scripts and performing all formal proofs interactively by using ProofPower (but still using Z Producer to translate Simulink), and so it is possible to compare the times taken. The speed up achieved by using ClawZ is impressive. The refinement script for a typical subroutine would take around one month to produce, and the verification conditions would take around one month to prove. Using ClawZ this is all done in about a day, and can mostly be performed by an analyst with considerably less expertise. This speed up of 50 times is not wholly representative, since there was probably considerable scope for improving the manual process. However, it gives an idea of the order-of-magnitude improvement that has taken place.

The sizes of the intermediate objects passed between the tools can vary considerably, but tend to be large. For a typical 80 line Ada subroutine, the refinement script outputted by RSG tends to range from 1,000 to 10,000 lines. Note that, in terms of numbers of lines, the witness script written by the analyst is around 10 to 50 times shorter than the resulting refinement script. Also note that the witness script is considerably more abstract than the refinement script, and so less susceptible to tedious errors in detail. Individual verification conditions tend to range from 100 to 1,000 lines of Z.

For the examples that can be compared, these objects tend to be around 4 times the size of those produced in the pre-ClawZ days. The reason for the increase in size is ultimately due to the systematic way in which RSG works, where automation is introduced at the expense of some conciseness. Raw refinement scripts and verification conditions are never seen by the analyst, and so conciseness is not an issue unless it impacts significantly on execution time.

The success of Supertac in being able to prove such huge verification conditions automatically is down to two main factors. The first is that the verification conditions are rarely as complicated as they might first seem, and much of the proof of a verification condition involves relatively trivial reductions. However, it would be insufficient for Supertac to only perform such trivial reductions, since the proofs of most verification conditions would still require some complex reasoning taking several hours to perform interactively.

The second main factor is that Supertac is designed with the context in which these verification conditions are generated in mind. Z Producer creates Z specifications that follow a well defined structure, computer programs for control systems tend to be written in similar ways, RSG generates refinement scripts in a highly systematic way and DAZ generates verification conditions according to a strict set of rules. It was possible to take all of these conventions into account when designing Supertac.

So, in summary, the architecture of ClawZ revolves around the classic formal verification toolset model of using a translator, a formal refinement tool and a theorem prover. The way that ClawZ differs is that it incorporates additional tools, in particular RSG and Supertac, that shield the analyst from tedious details normally associated with formal verification, and so enable analyst input and expertise to be dramatically reduced. These additional tools are able to achieve this by being dedicated to a specific verification domain.

3.2 Soundness

Formal verification tools are useful for providing positive evidence that software correctly meets its specification, as well as for finding errors. This positive evidence is very important if a toolset is being used for the certification of safety-critical software. Thus it is important that there is a high degree of confidence in the soundness of a formal verification toolset.

The success of ClawZ in being able to analyse software at relatively low cost is almost entirely due to the RSG and Supertac tools. However, equivalents of these tools do not exist in other formal verification toolsets, and they are implemented in several thousand of lines of source code. A concern might be that errors in RSG or Supertac may compromise the soundness of the toolset.

RSG, however, cannot introduce unsoundness to ClawZ. This is because all that RSG does, ultimately, is produce a refinement script. The way in which the refinement script was constructed does not affect soundness. If we assume that DAZ itself is sound, then it is impossible for anything, including RSG or the witnessing analyst or someone manually writing a refinement script, to fool it. DAZ will only produce verification conditions that are all provable if the initial specification statement in the refinement script is correctly implemented by the refined source code in the refinement script.

It might be possible to fool the analyst, however, by feeding DAZ a refinement script that has an initial specification statement that does not reflect the analyst's specification, or a refinement script that refines to source code different from the original. To give a pathological example, regardless of the analyst input in the specification and witnessing stages, DAZ could be presented with an initial specification statement that has a postcondition of “true” that is refined to a “null” Ada statement. This would result in all the verification conditions being provable, regardless of whether the original source code met its intended specification. However such situations cannot happen due to RSG, since it is

CPS, and not RSG, that inserts the initial specification statement, and since CPS checks that the Ada source code resulting from the refinement is syntactically (and thus semantically) equivalent to the original source code. Thus RSG, by the way that it is used, cannot introduce unsoundness to ClawZ.

Supertac, too, cannot affect the soundness of ClawZ, but for different reasons. Supertac is built on top of the ProofPower theorem prover, which is an LCF-style theorem prover [9]. A theorem prover is said to be sound if it is only possible to prove conjectures that are true. Great care is taken to make theorem provers sound. However, they are large programs that perform highly complex manipulations. Given the subtleties of mathematics, it is very difficult to ensure that nowhere in the large theorem prover program is a mistake that introduces unsoundness. There are numerous steps that can be taken to reduce the risk of this, but the most effective is to make the theorem prover LCF-style [10].

In the implementation of an LCF-style theorem prover, a special data type is reserved for theorems, i.e. conjectures that have been proved. Strong data typing of the programming language is then used to ensure that theorems can only be constructed from a kernel of operations, called *primitive inference rules.* New operations to prove conjectures can be defined outside the kernel, but they must ultimately be implemented in terms of the primitive inference rules. Thus an LCF-style theorem prover is sound so long as its primitive inference rules are sound and the small amount of source code implementing these is correct.

ProofPower's primitive inference rules are based on the HOL logic system [5]. This is the most widely used logic system used for LCF-style theorem provers, and is one of the simplest, and so it is perhaps the best logic system to use to minimise the risk of unsoundness. Thus ProofPower, being a HOL-based, LCF-style theorem prover, has a very high pedigree for soundness. Also, any new proof operations built on top of ProofPower cannot introduce unsoundness, because they must ultimately be defined in terms of primitive inference rules. So, since Supertac is built on top of ProofPower, which is an LCF-style theorem prover, it cannot introduce unsoundness.

Thus the additional tools incoporated into ClawZ, i.e. RSG and Supertac, that differentiate ClawZ from traditional formal verification toolsets and enable it to be used so much more cost-effectively, are incorporated in such a way that cannot compromise soundness of the overall toolset.

4 Conclusions

Control systems play an important role in modern society, and the increasing complexity of their software is causing the associated development, maintenance and certification costs to escalate. The ClawZ toolset is dedicated to the formal verification of control system software, and has been successfully applied to the industrial formal verification of Eurofighter Typhoon's FCC at much reduced certification costs when compared with other formal verification toolsets. ClawZ adapts the classic formal verification toolset architecture by including two novel components, RSG and Supertac, that are key to achieving its effectiveness.

RSG enables the analyst to construct a relatively abstract witness script instead of the refinement script that is required with existing formal verification toolsets. This enables the costly task of manual program refinement to be replaced by a much simpler, although still non-trivial, task. Supertac highly automates the formal proof, completing over 95% of verification condition proofs entirely automatically, and simplifying the remainder for interactive proof. This greatly reduces the costly task of interactive formal proof, and enables the witness script construction and interactive proof roles to be easily separated.

By being tailored to a specific verification domain, namely the verification of control system software specified in Simulink and implemented in Ada, these two components dramatically reduce the amount of analyst input and expertise required to perform formal verification. Furthermore, they are incorporated into the toolset in such a way that guarantees they cannot introduce unsoundness.

Although the toolset enables highly productive verification as it stands, there is still significant scope for improvement to help further reduce analyst time and expertise. Both RSG and Supertac could provide more feedback to the analyst to help in tracking down analyst errors more quickly. Also, Supertac could be enhanced to further automate verification condition proof.

There is nothing about this approach to building a formal verification toolset that limits it to this specific verification domain. Such an approach could equally be used to verify systems implemented in other programming languages, or specified in other design notations. Also, the scale of the cost reductions brought about by ClawZ opens up the prospect of much wider use of formal verification, including during development, and perhaps even for non safety-critical code. This approach could therefore have a profound impact on the use of formal verification in industrial applications.

References

1. The Mathworks Inc.: Using Simulink. 5th edn. (2002)
2. TA Consultancy Services Ltd.: MALPAS Training Course Notes. 2nd edn. (1995)
3. J.Woodcock, J.Davies: Using Z: Specification, Refinement and Proof. 1st edn. Prentice Hall (1996)
4. Lemma 1 Ltd. website: www.lemma-one.com.
5. M.J.C.Gordon, T.F.Melham, eds.: Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press (1993)
6. C.O'Halloran, A.Smith: Verification of Picture Generated Code. In: 14th IEEE ASE, IEEE Computer Society Press (1999)
7. C.O'Halloran, A.Smith: Don't Verify, Abstract! In: 13th IEEE ASE, IEEE Computer Society Press (1998)
8. C.Morgan: Programming from Specifications. 2nd edn. Prentice Hall (1994)
9. M.Gordon, A.Milner, C.Wadsworth: Edinburgh LCF – A Mechanical Logic of Computation. In: LNCS 78, Springer-Verlag (1979)
10. J.Harrison: Metatheory and Reflection in Theorem Proving: A Survey Critique. Technical report, University of Cambridge Computer Laboratory (1995)