ClawZ: Control laws in Z

R. Arthan
Lemma 1 Ltd, UK
rda@lemma-one.com

P. Caseley, C. O'Halloran, A. Smith
DERA Malvern, UK
{p.caseley, c.ohalloran, a.smith}@eris.dera.gov.uk

Abstract

ClawZ is a prototype tool whose objective is to link the Simulink® control engineering tool, from MathWorks, with the ProofPower® *dialect of Z. It provides a bridge between the use of Simulink to define control law diagrams and a tool to formally prove compliance between Ada and Z. The tool has been used as part of the formal proof of a Non-linear Dynamic Inversion flight control system comprising 37 pages of diagrams, 45 pages of Z and 1200 lines of non-comment Ada.*

1 Introduction

ClawZ (pronounced Claws) is a research prototype whose objective is to link the Simulink [1] system with the ProofPower [2] dialect of Z [3] and, in particular, to provide a bridge between the use of Simulink to define control law diagrams and the use of the Compliance Tool [4, 5] component of ProofPower to formally prove compliance between Ada programs and their Z specifications.

ClawZ operates by translating a Simulink model into a Z specification. This Z specification can then be used in conjunction with a library of supporting definitions to construct a Compliance Argument which can then be formally verified using ProofPower. Figure 1 illustrates the main inputs and outputs of this process. A rectangle denotes a file and the oval denotes a program.

The rest of this paper is structured as follows. Section 2 gives a more detailed technical overview of what ClawZ does based on a simple example model. Section 3 describes the use of the ClawZ translator to translate a Simulink model file into Z. Section 4 describes a complete compliance argument based on the example presented in section 2. Section 5 gives the proof scripts for the compliance argument given in section 4. Section 6 descibes how the approach can be extended to multirate control laws and control laws whose implementations are distributed and running concurrently. Section 7 describes the relationship to other work. Section 8 describes how ClawZ has scaled up to large control laws and the plans for the future.

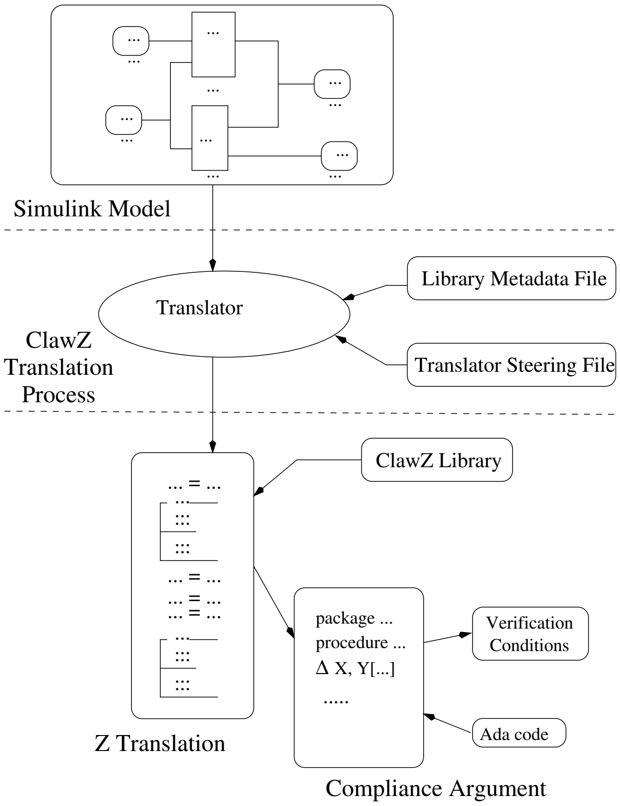

Figure 1. ClawZ Process

2 Overview of ClawZ

2.1 A simple Simulink model

To give an overview of the operation of ClawZ, it is helpful to give a simple example of a Simulink model. Figure 2 shows a model comprising an input port, *In1* and an output port *Out1* connected by a unit delay block, *DelayBuffer*.

This models a system which derives an output signal from an input signal. The input signal, I say, is sampled at regular time intervals[1] to give a sequence of discrete inputs $I_0, I_1, I_2, \ldots$. The output signal, O say, will then comprise the sequence $0, I_0, I_1, I_2, \ldots$. I.e., O is defined by $O_0 = 0$ and $O_i = I_{i-1}$ $(i > 0)$.

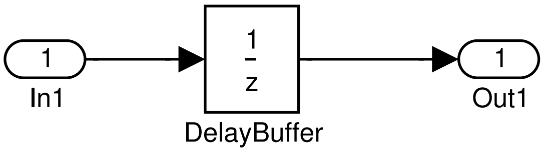

Figure 2. A Simple Simulink Model

Control engineers use z-transforms (not to be confused with the Z specification language) to transform a dynamic system modelled as function of time to a function of an algebraic variable z. Operations such as integration and differentiation then become simple multiplications by a function of z.

In the parlance of z-transforms, the unit delay operator of our simple model corresponds to multiplication by $\frac{1}{z}$, so Simulink labels the unit delay block in figure 2 with $\frac{1}{z}$. In a software implementation of the model, the unit delay corresponds to a buffer, so we have chosen to name the block $\mathit{DelayBuffer}$.

2.2 The model in Z

The purpose of ClawZ is to translate a Simulink model into a Z specification. The Z translation of the model of figure 2 is shown in figure 3. The model has resulted in two Z schemas.

The first schema corresponds to the library block, $\mathit{DelayBuffer}$. The name, $\mathit{unitdelay}$, of the model has been used as a prefix for the schema name to reflect the position of the block in the hierarchic structure of the model. The schema $\mathit{unitdelay\_DelayBuffer}$ is defined in terms of a library function $\mathit{UnitDelay\_g}$ with initial value zero. This library of Z definitions will be discussed in more detail in section 2.3 below.

The second schema $\mathit{unitdelay}$ represents the wiring of the diagram in figure 2. Its declaration part declares the three blocks which appear in the diagram (input port, unit delay block, and output port). The predicate part of the schema gives equations indicating that the output and input of the unit delay block are wired to the output port and input port respectively.

[1] This is a *discrete* model. Simulink also supports continuous models and hybrids. Currently, ClawZ is mainly applicable to discrete models.

z

$$\mathit{unitdelay\_DelayBuffer} \mathrel{\widehat{=}} \mathit{UnitDelay\_g}(X0 \mathrel{\widehat{=}} 0)$$

z

$\mathit{unitdelay}$

$$\mathit{In1?} : U;$$
$$\mathit{DelayBuffer} : \mathit{unitdelay\_DelayBuffer};$$
$$\mathit{Out1!} : U$$

$$\mathit{DelayBuffer.Out1!} = \mathit{Out1!};$$
$$\mathit{In1?} = \mathit{DelayBuffer.In1?}$$

Figure 3. The Simple Model in Z

2.3 Overview of the ClawZ library

As the simple example has shown, the semantics of Simulink library blocks like the unit delay block is carried into Z by a library of Z definitions. These definitions are typically generic functions, which when applied to appropriate arguments result in operation schemas following the usual Z conventions augmented with a special convention for handling initial values.

Figure 4 shows the definition of the library function $\mathit{UnitDelay\_g}$ that is used in the translation of our simple example. The function is generic with respect to the type, X, of the inputs and outputs of the unit delay block. In our example, $X = \mathbb{Z}$.

The function is parametrised by a Z binding giving the initial value of the state; applying the function to a particular binding, say $(X0 \mathrel{\widehat{=}} 0)$, as was done in figure 3, results in a schema equivalent to the one shown in figure 5.

As can be seen in figure 5, the ClawZ library functions adopt the convention of including the initial value of the state of a schema as a component in the schema. In our example, the initial value of 0 has been picked up from the "initial condition" parameter of the Simulink unit delay block in figure 2.

Z

$$
\begin{array}{l}
[X] \\
\hline
UnitDelay\_g : [X0 : X] \rightarrow \\
\quad \mathbb{P}\ [In1?,\ initial\_state,\ state,\ state',\ Out1! : X] \\
\hline
\forall\ pars : [X0 : X] \bullet \\
\quad UnitDelay\_g\ pars = \\
\quad\quad [In1?,\ initial\_state,\ state,\ state',\ Out1! : X\ | \\
\quad\quad\quad initial\_state = pars.X0\ \wedge \\
\quad\quad\quad Out1! = state\ \wedge \\
\quad\quad\quad state' = In1?]
\end{array}
$$

Figure 4. The Unit Delay Library Block in Z

Informal Z

$$
\begin{array}{l}
unitdelay\_DelayBuffer \\
\hline
In1?,\ initial\_state,\ state,\ state',\ Out1! : \mathbb{Z} \\
\hline
initial\_state = 0; \\
Out1! = state \\
state' = In1?
\end{array}
$$

Figure 5. The Unit Delay Library Block Instantiated

3 Using the translator

3.1 Files Used by the Translator

As suggested in figure 1, the ClawZ translation process involves four files. The first is the Simulink model file. This is the file produced when a Simulink model is saved and is the main input to the translation process. The suffix ".mdl" is normally used for this file, e.g., "unitdelay.mdl".

The second file involved in the translation process contains the Z translation of the Simulink model and is the output of ClawZ. This is the file where the translator writes the Z translation of the model. The output file is written in ProofPower ASCII format to make it easy to transport between systems. To convert it to the ProofPower extended character set, the program "`conv_extended`" can be used which is supplied with the ProofPower document preparation package.

The third file involved in the translation process is the library metadata file. This file contains a description of the library blocks supported by ClawZ in a format that allows the translator to select the most appropriate Z translation for a library block. The full Z definitions of the library blocks, such as $UnitDelay\_g$ in figure 4, are held in a separate file. If the model uses blocks that are not supported by the supplied library, the library can be extended by the user.

The fourth file involved in the translation process is the translator steering file. This file controls the way the translator maps Simulink names to Z identifiers. It is optional: if no translator steering file is provided, then the translator will use a default algorithm for the mapping. More information on this file is given in section 3.3 below.

3.2 ClawZ diagnostic output

When ClawZ translates a Simulink diagram it produces various error and/or warning messages.

The translation process has two main phases: a parsing phase when the model and other input files are read and checked and a semantic phase in which the model is analysed and translated. At the end of the semantic phase, the Z translation is written to the output file.

Errors during the parsing phase are generally fatal, although some warning messages may be generated. Typically a parsing error might indicate that the model had come from an incompatible version of Simulink or might arise from an error in the translator steering file or the library metadata file.

The translator attempts to recover from errors during the semantic phase. The most common error in this phase is when the translator encounters a usage of a library block that it cannot translate. In this circumstance, it generates an error message and continues, effectively ignoring the problematic block. This enables an assessment of the coverage of the supplied library for the model and a decision as to whether it will be feasible to extend the library to cater for the model.

The line number in the error message for an unsupported block refers to the subsystem containing the block, not the block itself, so generally a text editor will have to be used to search for the block. The block name and type is included in the error message.

3.3 Translator steering file

The translator steering file allows the translator's algorithm for mapping Simulink names into Z to be overridden. The format of this file is as shown in the following example:

```
NameMapping {
 UnitDelay            "Unit Delay"
 ZeroOrderHold        "Zero-Order\nHold"
}
```

Each line in the body of the file contains a Z name followed by a Simulink name (or list of same, separated by underscores, constituting a hierarchic name reflecting the position of a block within a model). The Simulink names must be enclosed in quotation marks. The idea is that if an association between a Z name and a Simulink name is specified in this file, the translator will use the association specified instead of its default algorithm for that Simulink name.

Right at the end of the translation process, the translator outputs a list of the Simulink names that it could not use directly as Z names. This list is in the same format as the translator steering file, so can be copied into a file and used as the basis for a translator steering file giving the preferred translation for these names (or any other names in the model).

4 Demonstrating compliance to the Z translation

In this section we give a Compliance Argument for an implementation of our simple example. This will formally argue that the implementation complies with the Z translation of the control law diagram.

Our plan is to implement the unit delay model using an Ada package along the following lines (in which, for simplicity, we have made the state variable global).

```
package UnitDelay is
  delay_buffer : integer;
  procedure step(sig_in : in integer;
                 sig_out : out integer);
end UnitDelay;
```

To relate this to the Z specification, we need an interface schema to relate the variables in the specification with the program variables. To construct the interface schema, we first of all define a schema declaring the relevant program variables. Note that the Compliance Tool normalises Ada names into upper case and uses a subscript 0 to distinguish the before-values of program variables.

z

UnitDelayProgVars

$DELAY\_BUFFER_0,\ DELAY\_BUFFER,$
$SIG\_IN,\ SIG\_OUT : INTEGER$

Now we define the interface schema which relates the input, output and state variables of the Z specification to their corresponding program variables.

z

UnitDelayInterface

$unitdelay;$
$UnitDelayProgVars$

$In1? = SIG\_IN;$
$Out1! = SIG\_OUT;$
$DelayBuffer.state = DELAY\_BUFFER_0;$
$DelayBuffer.state' = DELAY\_BUFFER$

We can now give the formal specification of the package, in which we existentially quantify over the schema $unitdelay$ so that only the program variables appear free in the post-condition. The list of variables after the Δ symbol are those variables whose values are allowed to change.

Compliance Notation

$package\ UnitDelay\ is$
$\quad delay\_buffer : integer;$
$\quad procedure\ step(sig\_in : in\ integer;$
$\qquad\qquad sig\_out : out\ integer)$
$\quad \Delta\ DELAY\_BUFFER,\ SIG\_OUT$
$\quad [\exists\ unitdelay \bullet UnitDelayInterface];$
$end\ UnitDelay;$

Finally, we give the implementation of the package, i.e. the package body. Note that the post-condition of procedure $step$ has been refined to a simpler form that just involves program variables, which in turn has been refined to a sequence of two Ada assignments. Verification Conditions (VCs) that demand these refinements are correct are generated by the Compliance Tool when the package body is processed.

Compliance Notation

$$
\begin{array}{l}
\textit{package body UnitDelay is} \\
\quad \textit{procedure step}(\textit{sig\_in} : \textit{in integer}; \\
\qquad\qquad\qquad \textit{sig\_out} : \textit{out integer}) \\
\quad \Delta\ \textit{DELAY\_BUFFER}, \textit{SIG\_OUT} \\
\quad [\textit{DELAY\_BUFFER} = \textit{SIG\_IN} \land \\
\quad\ \textit{SIG\_OUT} = \textit{DELAY\_BUFFER}_0] \\
\quad \textit{is} \\
\quad \textit{begin} \\
\qquad \textit{sig\_out} := \textit{delay\_buffer}; \\
\qquad \textit{delay\_buffer} := \textit{sig\_in}; \\
\quad \textit{end step}; \\
\textit{begin} \\
\quad \Delta\ \textit{DELAY\_BUFFER} \\
\quad [\exists\ \textit{unitdelay} \bullet \\
\qquad \textit{DelayBuffer.initial\_state} = \textit{DELAY\_BUFFER}] \\
\textit{end UnitDelay};
\end{array}
$$

Notice that a specification statement has been placed in the package initialisation demanding that the initial value of the program state variable is that specified in the Z specification. This specification statement is then refined to code which generates a further VC.

Compliance Notation

$$\sqsubseteq\ \textit{delay\_buffer} := 0;$$

The compliance argument is now complete. It generates four Verification Conditions (VCs). For completeness, the script to prove these VCs is given in section 5 below.

5 Proofs of the verification conditions

We now prove the VCs generated from the compliance argument above. First we use the Compliance Notation proof support tools to set up a proof context in which to work. This makes the Z translation of the control law diagram and the Z schemas declared during the compliance argument available for proof.

SML

```
all_cn_make_script_support"unitdelay_pc";
push_pc"unitdelay_pc";
```

Now we begin the proofs.

SML

```
set_goal(
[], get_conjecture"-""vcUNITDELAYbody_1");
```

ProofPower Output

$$
\begin{array}{l}
\textit{Now 1 goal on the main goal stack} \\
\\
(*\ ***\ \textit{Goal}\ ""\ ***\ *) \\
\\
(*\ ?\vdash\ *)\quad ⓩ\textit{true} \Rightarrow \textit{true}⌝
\end{array}
$$

This trivial VC arises from the empty preconditions and is proved just by *stripping*. This proof tactic includes removing universal quantifiers from the conclusion and moving antecedents into the assumptions. The conclusion is literally stripped of content, hence the name of the tactic. In this case the conclusion will become *true* and hence the VC is proved. The a is an abbreviation for $apply\_tactic$. The resulting theorem is then saved.

SML

```
a(REPEAT strip_tac);
save_pop_thm"vcUNITDELAYbody_1";
```

SML

```
set_goal(
[], get_conjecture"-""vcUNITDELAYbody_2");
```

ProofPower Output

$$
\begin{array}{l}
\textit{Now 1 goal on the main goal stack} \\
\\
(*\ ***\ \textit{Goal}\ ""\ ***\ *) \\
\\
(*\ ?\vdash\ *)\quad ⓩ\forall\ \textit{DELAY\_BUFFER}, \\
\qquad\qquad \textit{DELAY\_BUFFER}_0 : \textit{INTEGER}; \\
\qquad\qquad \textit{SIG\_IN} : \textit{INTEGER}; \\
\qquad\qquad \textit{SIG\_OUT} : \textit{INTEGER} \\
\qquad\quad \mid \textit{true} \land \textit{DELAY\_BUFFER} = \textit{SIG\_IN} \\
\qquad\qquad \land\ \textit{SIG\_OUT} = \textit{DELAY\_BUFFER}_0 \\
\qquad\quad \bullet \exists\ \textit{unitdelay} \bullet \textit{UnitDelayInterface}⌝
\end{array}
$$

This VC is ensuring that the program-oriented postcondition used for the procedure *step* in the package body meets the post-condition derived from the Simulink model as used in the package specification. We apply the usual simplifications for Compliance Notation VCs (which include rewriting with all applicable definitions in our present proof context) and then strip the goal. We then have an existential goal with an obvious witness: the only possible binding that meets the constraints of the interface schema. After supplying this witness, we have only to expand with the definition of $UnitDelay\_g$, which requires a little work because it is generic. The function $z\_gen\_pred\_elim$ is used to instantiate the generic definition of $UnitDelay\_g$ in section 2.3 with type $\mathbb{Z}$.

SML

```
a(cn_vc_simp_tac[] THEN REPEAT strip_tac);
a(z_∃_tac⌜(In1? ≙ SIG_IN,
          DelayBuffer ≙ (
              state ≙ DELAY_BUFFER₀,
              state′ ≙ DELAY_BUFFER,
              In1? ≙ SIG_IN,
              Out1! ≙ SIG_OUT,
              initial_state ≙ 0),
          Out1! ≙ SIG_OUT)⌝);
a(cn_vc_simp_tac[] THEN
  asm_rewrite_tac[z_gen_pred_elim[⌜U⊕ℙℤ⌝]
                  cn_UnitDelay_g_thm]);
save_pop_thm"vcUNITDELAYbody_2";
```

SML

```
set_goal(
[], get_conjecture"-""vcUNITDELAYbody_3");
```

ProofPower Output

```
Now 1 goal on the main goal stack

(* *** Goal "" *** *)

(* ?⊢ *)  ⌜∀ DELAY_BUFFER : INTEGER;
            SIG_IN : INTEGER
          • SIG_IN = SIG_IN ∧
            DELAY_BUFFER =
                DELAY_BUFFER⌝
```

This third VC is the one that ensures the body of the procedure *step* satisfies the post-condition in the procedure header. Stripping proves it immediately.

SML

```
a(REPEAT strip_tac);
save_pop_thm"vcUNITDELAYbody_3";
```

SML

```
set_goal([], get_conjecture"-""vc_1_1");
```

ProofPower Output

```
Now 1 goal on the main goal stack

(* *** Goal "" *** *)

(* ?⊢ *)  ⌜true ⇒
          (∃ unitdelay •
                DelayBuffer.initial_state = 0)⌝
```

This fourth and last VC is ensuring that the package initialisation code satisfies its post-condition. The proof is very similar to that of the second VC.

SML

```
a(cn_vc_simp_tac[]);
a(z_∃_tac⌜(In1? ≙ 0,
          DelayBuffer ≙ (
              state ≙ 0,
              state′ ≙ 0,
              In1? ≙ 0,
              Out1! ≙ 0,
              initial_state ≙ 0),
          Out1! ≙ 0)⌝);
a(rewrite_tac[z_gen_pred_elim[⌜U⊕ℙℤ⌝]
              cn_UnitDelay_g_thm]);
save_pop_thm"vc_1_1";
```

That completes the proofs for the unit delay example.

6 Extending the approach using CSP

The translation currently performed by the ClawZ tool is biased towards sequential code on a single processor. Indeed neither the SPARK language nor the compliance tool currently directly supports distributed applications or concurrency. Further the ClawZ tool assumes single rate, differing execution rates are not part of the Z model produced by the tool.

Unfortunately it is not unusual for a control law, for example in a flight control system, to be distributed over a number of microprocessors and communicate by means of variables held in a globally accessible memory. This leads to problems of race conditions in the computation of the control law, for example a shared global variable can be overwritten before it is read or stale data might be read. These problems are solved by a cyclic scheduler with a single global clock. This imposes the logical structure expressed in a control law diagram without losing advantages of concurrency. The problem now becomes how to verify the correctness of these schedulers with respect to the original diagram.

The approach taken by the Systems Assurance Group at DERA Malvern has been to combine the verification technology of the compliance tool with the verification technology of FDR for concurrent entities expressed in the language of Communicating Sequential Processes, CSP [6]. A CSP model has been developed that faithfully represents the flow of control through the individual boxes or subsystems of a control law diagram, even when parts are running at a different rate to the rest of the diagram. The functionality of the boxes or subsystems has been deliberately abstracted away. In the CSP model a box can only fire when its inputs, the outputs of previous boxes, are enabled by these previous boxes firing. The inputs to the whole diagram are assumed to be enabled. This model of the diagram acts as the specification against which an implementation can be verified.

The implementation is a set of subsystems that communicate synchronously due to the schedulers and a common view of time. The output from a subsystem can occur without synchronizing with any other output or input. The only synchronization that must occur is between schedulers on the different microprocessors with a 'tock' of the global clock. The CSP refinement checks the data flow in the distributed concurrent implementation is the same as the logical data flow in the original control law diagram.

The ClawZ tool will be extended to produce the descriptions necessary for the CSP models as well as the complementary Z descriptions of the functionality of the subsystems. The subsystems are implemented in Ada as scheduled procedure calls, each of these procedures are separately verified using the compliance tool. The distributed scheduling is verified using FDR, the refinement verification tool that employs model checking .

7 Relationship to other work

There are three possible approaches to solving this problem: follow a method of "correct by construction"; demonstrate that the implementation of the autocoder is correct; or demonstrate compliance between a control law representation and code generated from it for each critical application.

The correct by construction approach is being investigated by Lucas Aerospace under a contract from the UK MOD [7]. A functional specification representing the control law diagram is transformed using already proven laws into code. This is an ambitious approach which is equivalent to developing a refinement calculus for control law diagrams. It is not clear to what degree the transformations could be automated, but potentially it could provide a very useful tool for cost effective verified code generation. Unfortunately the commercial market is such that tools such as Simulink and MATRIX$_x$® [8] will be more widely adopted. This is because of the extra and improved facilities which they will provide on a regular basis in response to the market.

To demonstrate that an autocoder is correct to the same level of rigour as the correct by construction approach requires proof. This is an arduous task of the same level of difficulty as the correct by construction approach. This approach also suffers from the competition from COTS tools such as Simulink.

The final approach of demonstrating compliance between an individual control law and its implementation has been described in this paper and is at least an order of magnitude easier, but needs to be repeated for each control law. Tools such as Simulink also support automatic code generation. The advantage of tool generated code is that it does it in a consistent manner. This means that machine support can be developed which will ease the burden of proof, in this limited domain, quite considerably. The main advantage of the approach advocated is that it can be used with COTS tools such as Simulink and MATRIX$_x$.

8 Conclusions

The ClawZ tool has been validated against a significant flight control law supplied by the control theory group at DERA Bedford, UK. The control law consisted of 37 pages (each denoting a subsystem) and comprised of approximately 275 basic library control blocks. The whole control law diagram described the basic functionality needed to provide flight control for a whole aircraft. When the ClawZ tool was first run on this control law it was able to translate 90% of the diagram. The remaining 10% was later translated by updating the ClawZ library. Indeed the 1200 lines of Ada code implementing the diagrammatic specification was formally proven correct using the Compliance Tool. This required 60 person days of effort a dramatic reduction in effort, and therefore cost, over similar activities in the past.

The ClawZ library has now been sufficiently extended to translate the flight control law for the experimental VAAC (Vectored thrust Aircraft Advanced flight Control) Harrier. The VAAC Harrier control law is 10 times larger than the original control law supplied by DERA Bedford. These case studies demonstrate that the ClawZ tool is robust and scalable to realistic systems. It also demonstrates that the ClawZ library could be extended to include all the discrete library blocks of Simulink and even used in the Simulink manuals as a definition of their semantics. Indeed the complete ClawZ translation of a control law diagram can be used as its semantics.

Interestingly a small change to a control law diagram can result in a very different Simulink ".mdl" (model) file when saved. This file is an ASCII representation of the diagram which Simulink can read back in to re-draw the diagram. This makes a comparison of the two control laws (the original and slightly changed control law) using conventional differencing tools (tools which compare two files and report on their differences) almost impossible. However the ClawZ translation of the two control laws will reflect their diagrams and therefore differ only slightly where the actual functionality has changed. Differencing can therefore be performed using their Z translations.

The current ClawZ tool although robust and scalable is still limited to certain kinds of control systems. For example it cannot cope with multirate control systems. A multirate system contains blocks that are sampled at different rates, ClawZ assumes the sampling rate is identical throughout the system. The only other major issue facing ClawZ is that an implementation of the control law could be distributed and running concurrently. Both of these issues occur in real aerospace control systems. Fortunately the current research project is addressing both of these issues using CSP and the FDR refinement checking tool. A solution requiring the production of CSP, as well as Z, (a relatively small change in the ClawZ tool) is close at hand.

Further information about ClawZ can be obtained from URL http://www.lemma-one.com/clawz_docs/clawz_docs.html

References

[1] Simulink, *A tool for modeling, simulation and implementation of control systems*, see URL http://www.mathworks.com.

[2] ProofPower, *A tool for specifying and reasoning in Z*, available from Lemma 1 Ltd, c/o Interglossa, 2nd Floor, 31A Chain St, Reading, Berks RG1 2HX, UK, see URL http://www.lemma-one.demon.co.uk/ProofPower.

[3] J Davies & J Woodcock, *Using Z*, Prentice Hall series in computer science, 1996.

[4] C O'Halloran, R Arthan & D King, *Using a formal specification contractually*, Formal Aspects of Computing Journal, Springer, Vol 9, No 4, 1997.

[5] C O'Halloran & A Smith, *Verification of Picture-Generated Code*, Proceedings of the 14th IEEE International Automated Software Engineering Conference (ASE), IEEE Computer Society, 1999.

[6] A W Roscoe, *The Theory and Practice of Concurrency*, Prentice Hall series in computer science, 1998. Also see URL http://www.comlab.ox.ac.uk/oucl/publications/books /concurrency.

[7] P. Garbett et al, *Secure Synthesis of Code: A Process Improvement Experiment*, proceedings of FM99.

[8] MATRIX$_x$, *A tool for modeling, simulation and implementation of control systems*, see URL http://www.isi.com.