A Tool Chain for the Automatic Generation of *Circus* Specifications of Simulink Diagrams

Chris Marriott, Frank Zeyda, and Ana Cavalcanti

Department of Computer Science, University of York, UK
{chris.marriott,frank.zeyda,ana.cavalcanti}@cs.york.ac.uk

Abstract. Previous work described how to translate Simulink control law diagrams into *Circus* specifications to facilitate verification by refinement. This is not a trivial task; several tools have been developed to automate parts of the translation. This paper introduces a new tool chain that extends and integrates existing technology to cover the entire translation and cater for a larger set of diagrams. Our contributions include the integration of data types, generic definitions, and extension of the technique to model action and enabled subsystems. The tool chain has been validated using an industrial case study.

Key words: Z, CSP, ClawZ, control law diagrams, verification

1 Introduction

Control systems are commonly modelled using control law diagrams: a graphical notation with blocks and connecting wires. Each block represents a calculation or function, and can have state; wires connect inputs and outputs of blocks. Systems may be so complex that functionality is often defined in a number of separate diagrams, known as subsystems; these introduce a hierarchy.

As regulations for certification of safety-critical systems are being tightened, the use of formal methods is becoming increasingly encouraged. Various attempts have been made to express control law diagrams in formal languages [3, 6]. Particular attention has been given to diagrams in MATLABs Simulink [9], a *de facto* standard, especially in the automotive and avionics industries.

Circus [11] is a formal language capable of expressing state-rich concurrent systems based on Z [12], CSP [10], and a refinement calculus [5]. In [4], Cavalcanti *et al.* describe a formalised translation from Simulink diagrams to *Circus* models; it takes into account parallelism and independent flows of execution.

The main benefit of using *Circus* to encode Simulink diagrams is the ability to prove correctness of implementations through refinement. The work presented here extends the set of translatable diagrams, automates further the model generation technique of [4], and expands the set of programs we can prove correct. Code generation and diagram validation techniques that extend the static analysis in Simulink exist to satisfy different objectives from those we address here.

Our work extends and integrates a number of tools and associated techniques to support a single-click translation, and handle a larger set of diagrams.

As a result, users can produce *Circus* specifications without the need for indepth knowledge of multiple tools. We describe here the enhancements to existing tools and new methods that make this possible. In particular, we address the integration of data types and type-sensitive translation, the use of generic definitions, the use of a complex tool chain in the context of safety-critical systems, and techniques to model enabled and action subsystems.

Previously, there were two tools that supported the conversion of Simulink diagrams to *Circus* (namely ClawZ [2] and ClawCircus [13]). Each is driven individually with a significant amount of manual input. Expertise is required in Simulink, Z, *Circus*, and methods to bring these components together. With our tool chain, the amount of expert knowledge and manual input is reduced.

Additionally, the current technique does not cater for enabled and action subsystems. These are just like other subsystems in Simulink, which are defined by a sub-diagram, except they have enabling conditions that determine whether they are executed or not. They are used to control the flow of execution in a diagram and are commonly used in industrial applications. Here, we present a technique to model enabled and action subsystems in *Circus*.

The remainder of this paper is structured as follows. Section 2 presents preliminary material related to our work. Section 3 describes the translation from Simulink to *Circus*. Section 4 introduces enabled and action subsystems and describes how they can be expressed in *Circus*. Finally, Section 5 explains how the chain has been applied to a large industrial example not previously translatable, along with our conclusions and possible further work.

2 Background

This section describes Simulink diagrams, *Circus*, and existing tools.

Control law diagrams An example control law diagram written in Simulink notation can be seen in Figure 1. It specifies a missile guidance subsystem used in the aerospace industry [9]. Individual blocks are boxes on the diagram, and perform their own unique function; arrows between blocks represent the communication of values. The small ovals are the inputs and outputs to the subsystem (Rm, Vc, AZ_d, ...). This example also contains an enabled subsystem (*Fuze*).

The example is used to locate an initial target position and then monitor the flight of the missile using closed-loop tracking to ensure it is reached; these calculations are performed within the custom *Guidance Processor* subsystem. The *Fuze* subsystem is used to control the detonation of the missile; it monitors the distance to the target and feeds back into another tracking subsystem.

Circus language Systems are specified through processes in *Circus*. Features from Z and CSP are available, including schemas, communication, parallelism and choice. Programming operators come from Morgan's refinement calculus.

The main constructs are channels, processes and actions. Channels are used to define communication events between processes. Processes contain state information and have a behaviour defined by actions. State is local, so that interaction

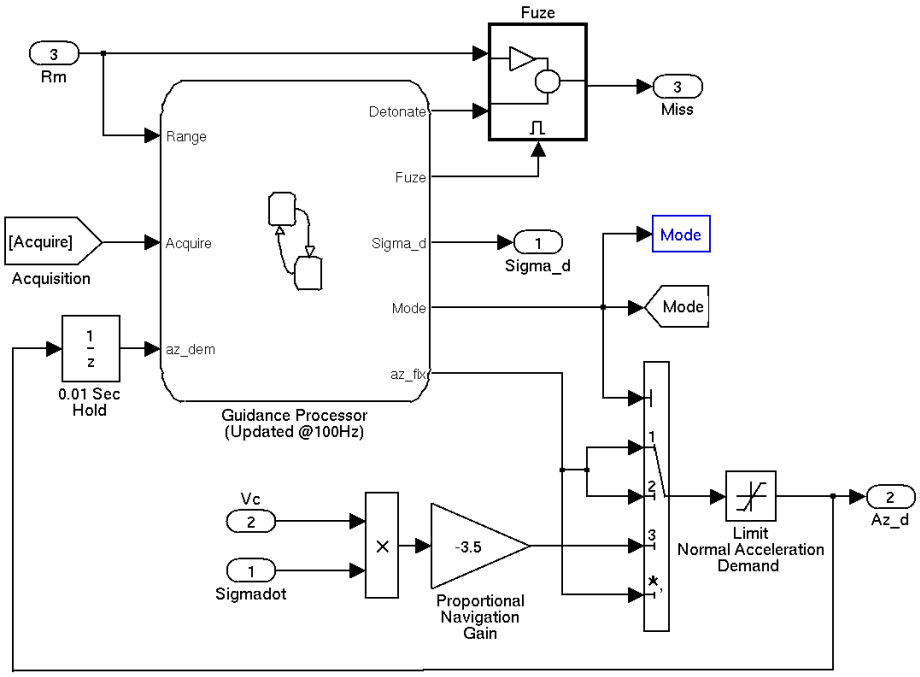

Fig. 1. Guidance Subsystem in Simulink [9]

can only occur through channels. A process can be defined explicitly, or by using operators of CSP for composition of other processes, such as parallelism.

An action can be defined as either a schema, which performs operations on the process state, a command in Dijkstra's guarded command language, or a CSP expression. Local actions are referenced by the main action, which specifies the behaviour of the process. More details about *Circus* can be found in [11].

ClawZ is a tool suite for verification of implementations of Simulink diagrams [2]. It translates diagrams into Z encoded for ProofPower-Z [7], a mechanical theorem prover. ClawZ has been used in industry and has reduced costs of verification [1].

In ClawZ models, schemas are used to define inputs, outputs, and state elements of blocks and subsystems. Only discrete-time blocks are translated because software is discrete. Schemas produced by ClawZ are defined in a library; attributes in diagrams are used to match blocks to corresponding library schemas.

Circus specifications use schemas defined by ClawZ to describe functionality; CSP describes the communication and behavioural aspects of the control law.

3 Translating Simulink diagrams into *Circus* specifications

This section describes our tool chain to translate Simulink diagrams into *Circus* specifications automatically. We describe all tools required and explain how

each was tailored or developed to achieve the integration in Figure 2; ovals represent files and libraries, and squares represent processes and tools. The dotted-borders indicate processes or tools adapted or developed to create the chain; the two shaded ovals are the Simulink input and *Circus* output files.

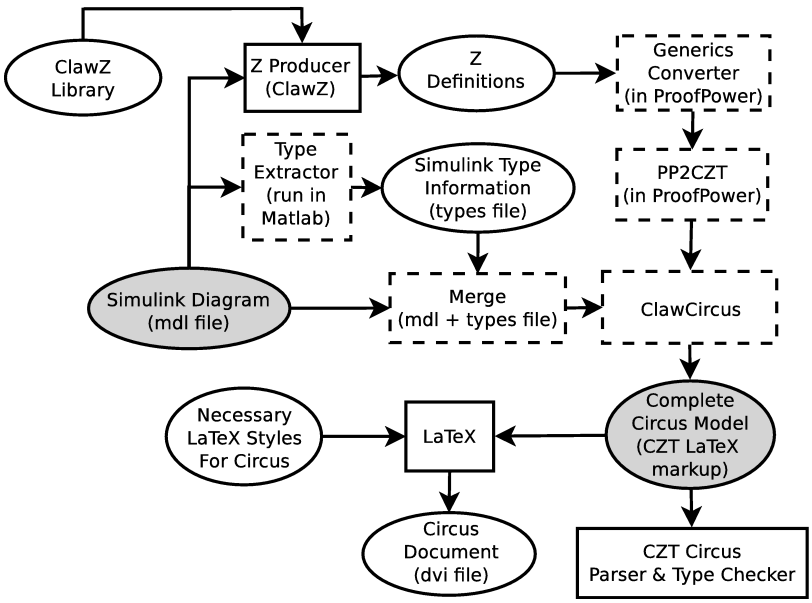

Fig. 2. Simulink to *Circus* Translation Process

3.1 Z Producer, Generics Converter, and PP2CZT

The three processes in the top path in Figure 2 are used to produce and modify the necessary Z definitions for use in the *Circus* model. The Z Producer (part of ClawZ), is used to specify schemas for blocks automatically.

The ClawZ output is encoded for ProofPower-Z and can be used for verification once parsed. A problem arises, however, because the use of generics in ProofPower is different from that in standard Z (and *Circus*). ProofPower-Z allows partial instantiation of a generic definition: it is possible to use generic definitions in ProofPower-Z without instantiating all formal generic parameters.

The ProofPower-Z notation includes the universal type $\mathbb{U}$, which is the carrier set of a generic type ($\mathbb{U}[X] \mathrel{\widehat=} X$). The Z definitions produced by ClawZ use the universal type because of the lack of type information in the Simulink file. Data types are inferred automatically within ProofPower-Z and this remains true for the majority of definitions when treated as part of a standard Z specification. In some cases, however, inference of actual generic parameters is not possible; as an example, we consider the definition below of a Selector block.

$$
\begin{array}{|l}
\hline
Selector \\
In1? : \mathbb{U} \\
Out1! : \mathbb{U} \\
\hline
Out1! = In1?\,(1) \\
\hline
\end{array}
$$

The Selector block takes a sequence of values ($In1?$) and selects a particular element ($Out1!$); in this example it is the first element. In ProofPower-Z, this schema is valid: it has an implicit formal generic parameter as there is not enough information to fully instantiate the type of $\mathbb{U}$. This parameter is not declared explicitly and the schema is not well typed according to the rules of standard Z.

To overcome this, we rewrite the definitions from ClawZ in standard Z using the new Generics Converter tool. The schema below has the same semantics as the previous example; it is a standard Z definition that introduces the type X as a formal generic parameter rather than using the universal type of ProofPower-Z. The conversion automatically infers, from the schema in the ClawZ output, that the type of the input ($In1?$) is a sequence of values. This is represented as a relation from an integer to a value of the generic type parameter.

$$
\begin{array}{|l}
\hline
Selector[X] \\
In1? : \mathbb{Z} \leftrightarrow X \\
Out1! : X \\
\hline
Out1! = In1?\,(1) \\
\hline
\end{array}
$$

The Generic Converter traverses all definitions stored in a ProofPower-Z file and analyses their components to establish the type of definition and whether any implicit generic parameters exist. If none are found, the definition remains unchanged; however, upon finding generic parameters, the definition is re-constructed. The new definition contains the formal generic parameters explicitly.

The modified ClawZ output in standard Z is converted into CZT markup (used by the *Circus* parser) using the new PP2CZT tool within ProofPower-Z. It performs a syntactic translation of all definitions and schemas in the ClawZ output and automatically produces the CZT encoding. The translation relies on a set of mappings from the internal representation in ProofPower-Z to the text-based markup in CZT. All definitions in a ProofPower-Z document are considered individually; every component in the definition is then analysed, translated, and re-assembled in a new file to form the corresponding CZT definition.

3.2 Type Extractor and Merge

Blocks in Simulink have a set of input and output ports, each with a specific data type and dimension. Previously, the translation assumed that all components were one-dimensional, and used the ProofPower-Z $\mathbb{R}$ data type to define their types. This, however, is not a realistic assumption, and since data types in Simulink are different to those of *Circus*, a mapping between data types is necessary. Simulink uses data types such as *double, int8* and *uint8;* we represent these

in *Circus* as $\mathbb{R}$, $\mathbb{Z}$, and $\mathbb{N}$. Simulink uses multi-dimensional data such as vectors and matrices, these are represented as sequences: $seq\ X$ for vectors and $seq\ seq\ X$ for matrices. Our extension also translates boolean and complex values, and is easily extended to include custom data types.

A challenge in achieving this translation was the fact that data types and dimensions are not recorded in the Simulink (*mdl*) file. We extract them using the new Type Extractor tool. This takes the *mdl* file and produces a *types* file containing data types and dimensions for all block inputs and outputs.

This is achieved by running a custom function within MATLAB; by executing inside the MATLAB environment, we can extract attributes of diagrams not stored in the *mdl* file. This tool iterates through all blocks in the diagram and produces a new file with the same structure as the original *mdl* file.

The extracted type information is combined with the original file by our new Merge tool. The purpose of Merge is to combine two *mdl* files into one *mdlx* file. A new file is created to maintain traceability and ensure the original diagram can still be used in Simulink. The Merge tool scans both input files for matching elements in the tree structure of systems, blocks, and subsystems. Attributes from matching pairs in both *mdl* and *types* files are merged in the new *mdlx* file.

3.3 ClawCircus

The majority of the translation is achieved using the ClawCircus tool, which takes the extended Simulink file and ClawZ output and produces a *Circus* specification. A description of the tool and its implementation can be found in [13].

What we needed to do to incorporate ClawCircus in the tool chain (apart from fixing a few bugs) was to provide a way of driving it without a graphical interface. Our new ClawCircus uses a configuration file to determine its input diagram and the required translation. This is useful in the safety-critical industry to ensure traceability; all graphical interfaces are removed in our chain.

The configuration file describes which part of a Simulink diagram to translate; the requested output could be a single block, a subsystem, or the entire system. It also defines whether a subsystem is expanded or collapsed. When expanded, the translation models all internal blocks as individual processes and combines them in parallel. When collapsed, the translation does not combine the blocks in parallel, but produces a centralised single process. It also defines whether the model is simplified or not. Simplified specifications do not contain vacuous definitions to ease readability, like empty schemas or actions without behaviour. Unsimplified versions have a more uniform structure; this is useful for automation of refinement where the shape of models is important.

Configuration files are simple and do not require additional expertise to produce. A parser to interpret the configuration file is now part of ClawCircus.

3.4 LaTeX, and the *Circus* parser and type checker

The *Circus* file produced is encoded in the *Circus* LaTeX markup and can be transformed into a viewable document; the type-set output makes it easier to

read. The tool chain produces two outputs: one in *Circus* LaTeX markup for the parser and type checker (Complete *Circus* Model), and a *dvi* file with the correct graphical notation for the specification (*Circus* Document).

The *Circus* Parser and Type Checker is invoked automatically to check the validity of the *Circus* specification generated. This does not validate the diagram per se, but provides some empirical evidence for the validity of the models produced by the translation. Additional tool support to analyse and refine *Circus* specifications is under development; ease of model generation crucially paves the way for those techniques to be applied effectively. Further validation of the models themselves comes from the fact that they have been used as a basis for a refinement technique formalise in [4] to verify control systems.

In summary, the tool chain eliminates the need for vast amounts of manual input and specialised knowledge. By combining a Simulink file with the corresponding configuration file, all output files are produced automatically. The specification is automatically passed through the parser and type checker with a detailed account of the entire process stored in a log file. The tool chain is automated using a script to invoke tools and manipulate files.

4 Enabled and action subsystems

This section describes translation enhancements to model enabled and action subsystems. Outputs from these subsystems depend on an enabling condition, which is determined by the value received on an enabling or action port. Enabled subsystems check if a value is greater than zero before being enabled, whilst action subsystems use a signal from a separate if-then-else or switch statement.

We consider, for example, the very simple diagram in Figure 3; it demonstrates the use of action subsystems, but is not the limit of our approach. In the example, an If Else block is used to control two action subsystems, which each have their own output. The If Else block takes an input (In1) and compares the value against some condition; in this example, the value must be greater than 4. If true, the If Else block outputs a boolean true to the first subsystem, and false to the other. If false, the boolean outputs to the subsystems are false and true respectively. The values from both subsystems depend on the enabling conditions; these are determined by the boolean value from the If Else block.

Both action subsystems output a constant value with a delay of one time unit. They also contain a block labelled Action Port, which is the boolean input signal from the If Else block and determines whether the subsystem is enabled.

Output blocks in enabled and action subsystems output a value, whether the subsystem is enabled or disabled; this value depends on the behaviour of the internal blocks. Typically, outputs from several subsystems are combined using a Merge block to ensure that only the value from the currently enabled subsystem is used. Subsystem outputs, however, may also be used individually, in which case, the output when enabled and disabled needs to be considered.

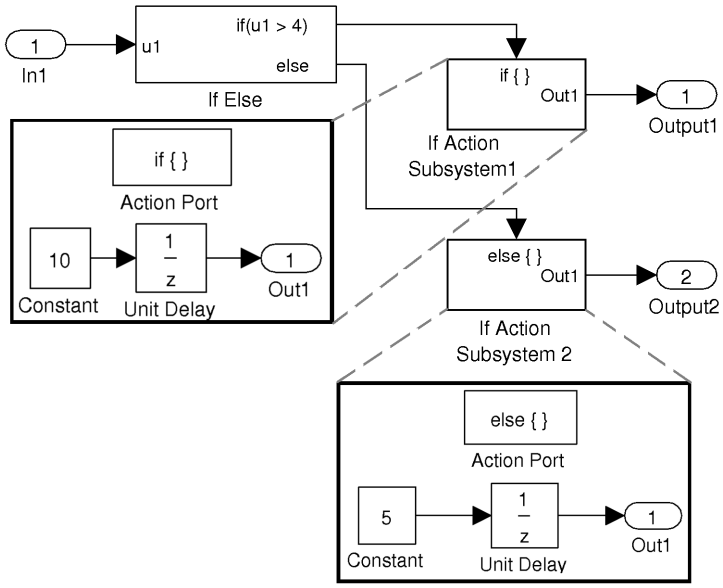

Fig. 3. Example If-Then-Else System

Additionally, by considering subsystems separately, rather than their combined use with other blocks, we obtain a compositional translation strategy.

All blocks inside these subsystems are essentially paused when disabled. Current values must be recorded along with the enabling condition for all blocks with state inside the subsystem. It is not sufficient to use the existing behavioural definitions from ClawZ for blocks with state, as they do not include the additional components required to capture the enabling/disabling behaviour. Also, output blocks require an additional field to represent their initial value when inside an enabled or action subsystem; this is the initial output value of the subsystem.

When subsystems are enabled, blocks behave as they would normally. When disabled, blocks with state and output blocks have additional properties that describe what to do with the output value: whether to hold the last value stored, or reset the output to the initial value. Additionally, when the subsystem is re-enabled, having been in a disabled state, both enabled and action subsystems can be configured to preserve the states of all internal blocks, or reset them to their initial state. This affects blocks that define an output sequence for example, which can either pause at the last value, or reset to the first value in the sequence. The configuration is static as the held and reset properties are defined within the Simulink diagram. In summary, we have four configurations of subsystems and their blocks that give rise to different behaviours as shown in Table 1.

The remainder of this section describes how we represent the subsystems and their internal blocks in *Circus*, with the necessary Z definitions.

Subsystem config.	Block config.	Block output when disabled	Subsystem state when re-enabled
Held	Held	Retains previous value	Retains previous state
Held	Reset	Set to initial value	Retains previous state
Reset	Held	Retains previous value	Internal block states are reset
Reset	Reset	Set to initial value	Internal block states are reset

Table 1. Enabled/action subsystem state and output combinations

4.1 Z definitions

The Z definitions used in the *Circus* model of a diagram have to be augmented to support enabled and action subsystems. This applies to blocks with state, as it is the state that is updated in different ways; to capture this we include three schemas for each block with state. These schemas describe the standard behaviour of the block, the behaviour when held, and the behaviour when reset.

As an example, we consider the Unit Delay block (as seen in Figure 1), which takes an input value, stores it in the current state, and outputs the value from the previous state; it is a single one place buffer. ClawZ uses a generic definition as it is applicable to many data types; the standard behaviour is as follows.

$$
\begin{array}{|l}
\hline
UnitDelay[X] \\
In1? : X;\ Out1! : X \\
initial\_state, state, state' : X \\
\hline
Out1! = state \land state' = In1? \\
\hline
\end{array}
$$

Consider now the situation where the Unit Delay block is inside an action or enabled subsystem and is disabled; the output is either held or reset. The ClawZ schema to describe the behaviour when held is below. The difference between this and the standard schema is in the value stored in the $state'$ component.

$$
\begin{array}{|l}
\hline
UnitDelay\_h[X] \\
In1? : X;\ Out1! : X \\
initial\_state, state, state' : X \\
\hline
Out1! = state \land state' = state \\
\hline
\end{array}
$$

The schema for the behaviour when reset, shown below, is different to the standard one as the components $state'$ and $initial\_state$ are defined to be the same.

$$
\begin{array}{|l}
\hline
UnitDelay\_r[X] \\
In1? : X;\ Out1! : X \\
initial\_state, state, state' : X \\
\hline
Out1! = state \land state' = initial\_value \\
\hline
\end{array}
$$

These three schemas define the behaviours of the block when inside an enabled or action subsystem, however, they are not sufficient for the *Circus* model. We require additional schemas that capture the value of the current and previous enabling condition. These additional components are crucial in order to define the four scenarios in Table 1. Firstly, we define a state schema which contains a single boolean value to record the enabling condition of the block.

$$\begin{array}{|l} \textit{Enabled\_State} \\ \hline enabled : \mathbb{B} \\ \hline \end{array}$$

This state component must be updated in accordance with the current enabling condition of the subsystem. Firstly we define a frame schema for the update operation that takes a boolean input and assigns it to the *enabled* state component.

$$\begin{array}{|l} \textit{Enabled\_Frame} \\ \hline \Delta \textit{Enabled\_State}; \\ Enabled? : \mathbb{B} \\ \hline enabled' = Enabled? \\ \hline \end{array}$$

We define three further schemas to capture the scenarios where the subsystem becomes enabled, remains enabled, and is disabled.

$$Enabling == [Enabled\_Frame \mid enabled = \mathbf{False} \land enabled' = \mathbf{True}]$$

$$RemainEnabled == [Enabled\_Frame \mid enabled = enabled' = \mathbf{True}]$$

$$Disabled == [Enabled\_Frame \mid enabled' = \mathbf{False}]$$

Using these three schemas to capture the enabling condition of a block in conjunction with the existing ClawZ block library definitions, it is possible to define block schemas for each of the four kinds of subsystem configuration in Table 1. Firstly, in the scenario where both the block and subsystem are set to hold their values when disabled and on re-enabling, we use a definition like that shown below for our example Unit Delay block. Both the *Enabling* and *RemainEnabled* schemas are combined with the *UnitDelay* schema that defines the normal behaviour. This is because when both enabled and upon re-enabling, the block values remain the same and normal behaviour continues. When the block is disabled, the *UnitDelay_h* schema is used as this specifies the held behaviour.

$$\begin{aligned} UnitDelay\_Augmented == &(Enabling \land UnitDelay) \lor \\ &(RemainEnabled \land UnitDelay) \lor (Disabled \land UnitDelay\_h) \end{aligned}$$

The second scenario is when the subsystem is set to hold the internal states upon re-enabling, and the block is set to reset to its initial value when disabled. The difference here is the *Disabled* schema is combined with the reset schema.

$$\begin{aligned} UnitDelay\_Augmented == &(Enabling \land UnitDelay) \lor \\ &(RemainEnabled \land UnitDelay) \lor (Disabled \land UnitDelay\_r) \end{aligned}$$

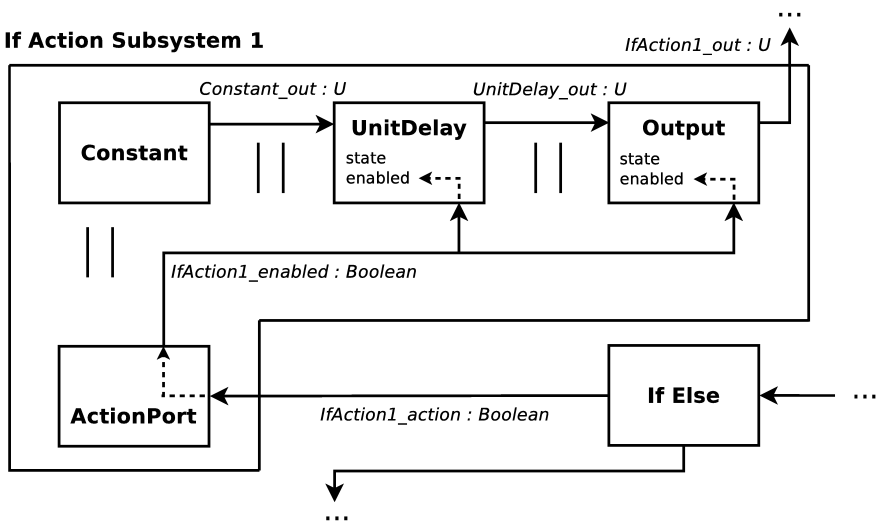

Fig. 4. *Circus* model for example action subsystem

The third behaviour is found when the subsystem resets the internal states upon re-enabling and the block holds its value when disabled. The *Enabling* schema is combined with the reset schema for the Unit Delay whilst the *Disabled* schema is associated with the held schema. Finally, the scenario where the block and subsystem reset their values. The $UnitDelay\_r$ schema is combined with both the *Enabling* and *Disabled* schemas.
The translation produces one of the four definitions above for each of the blocks with state inside an enabled or action subsystem, based on the properties of the subsystem and block.

4.2 *Circus* model

This section describes *Circus* processes that model enabled and action subsystems using the Z definitions presented above. As described previously, the translation of subsystems can be done in two ways. Firstly, all blocks can be translated individually and combined using parallel composition. Alternatively, the subsystem can be defined in one centralised process. (Semantically the models are the same, however, parallelism facilitates refinement to concurrent implementations).

With a centralised process, the Z definition from ClawZ that represents the overall subsystem is lifted into a *Circus* process. This definition includes instances of the schemas for each block in the subsystem and connects the inputs and outputs together just like in the original approach. The *Action*? input to the subsystem is connected to all of the *Enabled*? inputs of the blocks. The *state* and *enabled* conditions for all blocks in the subsystem are defined as state components in the *Circus* process to ensure information is not lost between invocations.

Translations that use parallel composition of *Circus* processes for blocks in the subsystem are slightly different. To represent the *Action*? input to the subsystem, an additional *Circus* process is defined to pass on the enabling condition to the other blocks in the subsystem via a broadcast channel.

As a simple example, Figure 4 depicts the structure of the corresponding *Circus* model for the first subsystem in Figure 3. There, arrows represent synchronisation channels corresponding to wires in a diagram, whilst the two vertical bars inbetween processes indicate parallel composition. If Action Subsystem 1 and If Else are separate processes (which are composed in parallel to define the model of the complete diagram). The process If Action Subsystem 1 is itself defined by a parallel composition of four processes: Constant, Unit Delay, and Output correspond to the blocks in the diagram, and Action Port is the extra process defined below. The channels *Constant_out* and *UnitDelay_out* correspond to the wires. The *IfAction*1*_enabled* channel broadcasts the enabling condition received on *IfAction*1*_action* from the If Else block; all internal blocks in the subsystem synchronise on this enabling signal.

The *Circus* process *ActionPort* is below; it operates in parallel with the other processes. The *end_cycle* channel is used as a synchronisation point for all parallel processes; only once all processes have synchronised on the *end_cycle* channel can each individual process recurse or terminate accordingly.

$$\textbf{process}\ ActionPort \mathrel{\widehat=} \mu X \bullet$$
$$IfAction1\_action?x \longrightarrow IfAction1\_enabled!x \longrightarrow \textbf{Skip}\ ;\ end\_cycle \longrightarrow X$$

Processes that represent blocks inside enabled and action subsystems cannot use the standard translation with our additional state components. Most significantly, processes have to synchronise on the channel that passes the enabling condition of the enabled or action subsystem to the blocks. We extend the state of blocks with the *Enabled* flag and relate this to the underlying ClawZ schema. In our example, the *Enabled*? value is taken from the *IfAction*1*_enabled* channel.

The *Circus* model for enabled subsystems differs slightly to the action subsystem example as the enabling input is not a boolean value, but either a scalar or vector value. The *EnablingPort* process pushes the boolean value true onto the enabled channel if any input value is greater than zero, and false otherwise.

Our approach extends the existing ClawZ and *Circus* model in a uniform and structured way, and lends itself to automation. A text-based algorithm is shown in Figure 5 to demonstrate the steps required to implement the translation.

5 Conclusions and further work

In this paper, we address several problems in translating Simulink diagrams to *Circus* and discuss modifications and extensions to existing tools to provide an automated solution via a tool chain. A more comprehensive description of all the details discussed in Sections 2 and 3 can be found in [8].

The individual stages of the translation shown in Figure 2 have been adapted and combined to automate the process. The only part of the translation not

1. For all blocks inside an action or enabled subsystem (apart from output blocks), check to see if the original ClawZ definition includes a state component. If no state exists, complete the block translation as in the previous technique.
2. For output blocks, and blocks with state, the additional frame schema described here must be combined with the original ClawZ schema according to the held and reset values of both the subsystem and the individual block.
3. Once all internal blocks are translated, the subsystem process is created:
   (a) If a centralised translation is required, instances of all internal blocks are included in the subsystem definition and are connected as per the wires in the diagram; the enabling condition is simply a component of the subsystem and accessed directly by the block schemas - there is no channel synchronisation. The state components of internal blocks are lifted to the state of the subsystem. The main action of the subsystem is a parallel composition defining the functional behaviour and the state update procedure.
   (b) A parallel translation creates individual processes for all blocks in the subsystem including the enabling/action port; the communication between processes is through channels as per the wires in the diagram. The overall subsystem process is constructed as the parallel execution of all processes that synchronise on the enabling condition and the *end_cycle* channel. The final step is to hide the internal workings of the subsystem process from the rest of the system; this is achieved by hiding all of the internal channels, leaving only the inputs and outputs of the subsystem visible.

Fig. 5. Algorithm to translate enabled and action subsystems

successfully integrated in the process is ClawZ; this is due to the high level of customisation required from the user to successfully produce a ClawZ output.

The tool chain has been applied to large industrial examples, in particular, a previously non-translatable Non-linear Dynamic Inversion controller provided by QinetiQ. This non-trivial example includes nested subsystems, generic definitions, and a range of data types. The translation equates to 38,000 lines of *Circus* and completes automatically with no errors. The example presented minor bugs in tools that had not been tested with such large examples previously.

Several other examples have been used throughout the development and testing phase to ensure specific modifications and extensions are correct. These tests are small in comparison to the larger example above, however, each is challenging in its own right to test a particular part of the translation. The tool is available, with an example, from `https://svn.cs.york.ac.uk/anonsvn/clawcircus`.

As an alternative to our approach, Caspi *et al.* use the formal language Lustre to represent Simulink diagrams [3]. A tool automates their translation from Simulink to Lustre, and from Lustre to source code using the Lustre C code generator. This technique is focused on the generation of implementations with a certified code generator and has proven popular in industry. Consider, however, the situation in which the code generation technique changes; the revised generator must be re-certified. This is an expensive and time consuming process;

should our technique to generate implementations change, the effort required to prove a modified refinement law is significantly less.

Chen *et al.* present a formal semantics and tool support to reason about functional and timing aspects of Simulink diagrams [6]. Their work presents a comprehensive library of translatable blocks for both discrete and continuous time. The work is focused on the validation of diagrams with the use of the PVS theorem prover; it does not address our larger interest in program verification.

Our translation function, previously defined in [4], is specified in a compositional manner and allows us to produce an individual *Circus* process for each block or subsystem in a Simulink diagram. As *Circus* has a semantics that supports compositional refinement, piecewise development is well supported.

Future work will mechanise the translation of enabled and action subsystems based on the algorithm described. Automation of refinement techniques will allow automatic generation of models of Ada programs for verification of implementations. Work is also ongoing to integrate time-specific Simulink diagrams in *Circus* using *Circus Time*; this work will further increase the set of translatable Simulink diagrams and make the tool chain applicable to more applications.

References

1. M.M. Adams and P.B. Clayton. ClawZ: Cost-effective formal verification for control systems. *Formal Methods and Software Engineering*, pages 465–479, 2005.
2. R. Arthan, P. Caseley, C. O'Halloran, and A. Smith. ClawZ: Control laws in Z. In *ICFEM*, page 169. Published by the IEEE Computer Society, 2000.
3. P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating discrete-time Simulink to Lustre. In *Embedded Software*, pages 84–99. Springer, 2003.
4. A. Cavalcanti, P. Clayton, and C. O'Halloran. From Control Law Diagrams to Ada via *Circus*. *Formal Aspects of Computing*, 2010. accepted for publication.
5. A. Cavalcanti, A. Sampaio, and J. Woodcock. A refinement strategy for *Circus*. *Formal Aspects of Computing*, 15(2):146–181, 2003.
6. C. Chen, J.S. Dong, and J. Sun. A formal framework for modelling and validating Simulink diagrams. *Formal Aspects of Computing*, 21(5):451–483, 2009.
7. D.J. King, R.D. Arthan, and I.C.L. Winnersh. Development of practical verification tools. *ICL Systems Journal*, 11:106–122, 1996.
8. C. Marriott. A Tool Chain for the Automatic Generation of *Circus* Specifications from Control Law Diagrams. Masters project thesis, Department of Computer Science, The University of York, 2010.
9. The MathWorks,Inc. *Simulink*. `http://www.mathworks.com/products/simulink`.
10. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
11. J. Woodcock and A. Cavalcanti. The Semantics of *Circus*. In *ZB 2002:Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer Berlin / Heidelberg, 2002.
12. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.
13. F. Zeyda and A. Cavalcanti. Mechanised Translation of Control Law Diagrams into *Circus*. In *Integrated Formal Methods*, volume 5423 of *Lecture Notes in Computer Science*, pages 151–166. Springer Berlin / Heidelberg, 2009.