Mechanised Translation of Control Law Diagrams into *Circus*

Frank Zeyda and Ana Cavalcanti

February 10, 2009

Abstract

Previously we proposed a strategy for translating control law diagrams into *Circus*. Combining elements from Z, CSP, and a refinement calculus, *Circus* captures functional and dynamic aspects of a diagram, and allows us to formally verify implementations. The main contributions of this paper are first to discuss a generalisation of the existing translation strategy, motivated by its mechanisation and application to sizable examples. Secondly, we present a tool, the *Circus* Producer, which automates the translation, and describe how its architecture facilitates subsequent development of further verification tools.

Keywords: Simulink, verification, ClawZ, Z, CSP

1 Introduction

Control law diagrams are commonly used by engineers in the specification and design of control systems. They describe a system as a directed graph of blocks carrying out elementary functions, and interconnecting wires transmitting data values between the outputs and inputs of the blocks. The diagrams can be structured in that (subsystem) blocks at a higher level can be defined in terms of subordinate diagrams at a lower level. The outputs of a diagram are repetitively computed in cycles of execution where inputs are taken and outputs calculated.

Where control law diagrams are used in the context of safety-critical systems, methods for analysis and validation are vital. Simulink [19] is a *de facto* standard for specifying control law diagrams and offers support for static analysis and simulation. Approaches based on model-checking have also been successfully used to verify properties of discrete-time and hybrid systems [18, 10, 11].

Most existing work indeed focuses on validating properties of diagrams; complementary to this, our concern is to verify the correctness of implementations. For this purpose, the ClawZ suite of tools [2, 1] has been developed and successfully used in industry. ClawZ verifies implementations by constructing a functional Z model of the diagram, and deriving a refinement conjecture for a given implementation. The implementation is typically written in a subset of Ada. Discharging the refinement conjecture is achieved in ProofPower-Z, a mechanical theorem prover for the Z language; it is performed mostly automatically.

A restriction of ClawZ is that it ignores the potential parallelism between the blocks of a diagram. In principle, the computations they define can be performed in parallel, with order imposed only by the way in which they are wired. Parallelism also surfaces when independent flows of execution within subsystems give rise to the possibility of outputs being produced before all inputs are received. Even some basic blocks, such as the UnitDelay, which delays a signal by one cycle, may produce their output prior to receiving their input.

To capture this aspect of control laws, we proposed an alternative technique based on *Circus* [6], a language for refinement that incorporates elements of Z, CSP, Dijkstra's Guarded Command Language and Morgan's refinement calculus [20, 7]. It is suitable for development of state-rich, reactive systems [14, 15]. In our *Circus* model of Simulink diagrams, we enrich the Z model produced by ClawZ to capture parallelism. The success of ClawZ in the avionics sector to reduce the cost of verification [1] strengthens our claim that a similar approach with equal benefit can be realised using *Circus* as a formal description language.

Due to the complexity and size of diagrams in real applications, tool support is indispensable to effectively generate the models that we propose. The main contribution of this paper is to report on a tool that mechanises the translation of discrete-time[1] single-rate Simulink diagrams into *Circus* specifications.

A further contribution is a generalisation of the translation strategy as presented in [6]. We obtain (a) more flexibility in defining the structure of the *Circus* models to match that of the proposed implementations and thereby facilitate the verification, (b) cover more sophisticated wiring, that is, those in which diagram outputs may refer to the same wire; and (c) simplify the interplay between Z and *Circus* to minimise the risk of introducing errors in the Z model.

In Section 2 we present ClawZ and give an overview of *Circus* and its verification technique for control systems; we also discuss how our tool integrates with the ClawZ framework. In Section 3 we explain the extensions that we propose to the translation strategy. Section 4 then describes the use of our translation tool, the *Circus* Producer, and Section 5 addresses some design and implementation issues. Finally, in Section 6 we draw our conclusions and address future work.

2 ClawZ and *Circus*

The verification process supported by ClawZ, and its major components are depicted in Fig. 1. First, the Simulink model is submitted to the Z Producer which generates the Z model encoded in the notation of ProofPower-Z. Each block or subsystem is defined by a schema introducing variables for the inputs, outputs and state components of the block. The set of bindings of the schema

[1]Discrete time is a requirement for diagrams to be implementable in software.

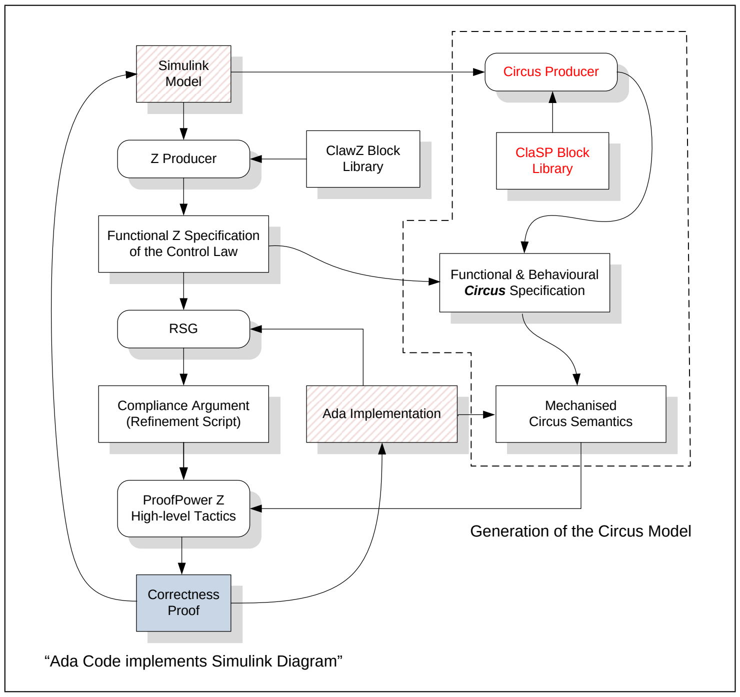
Simulink Model
Circus Producer
Z Producer
ClawZ Block Library
ClaSP Block Library
Functional Z Specification of the Control Law
Functional & Behavioural *Circus* Specification
RSG
Compliance Argument (Refinement Script)
Ada Implementation
Mechanised Circus Semantics
ProofPower Z High-level Tactics
Generation of the Circus Model
Correctness Proof
"Ada Code implements Simulink Diagram"

Fig. 1: ClawZ tools framework and integration of the *Circus* Producer.

specifies the behaviour of the block. The schemas for subsystems are dynamically constructed upon translation, but those for primitive blocks are inferred from a predefined library that may be extended by the user.

From the Z specification and the Ada implementation the RSG tool (Refinement Script Generator) constructs a compliance argument: a series of refinement conjectures which, if proved valid, establish the correctness of the implementation. Using ProofPower-Z and specialised proof tactics, these conjectures are proved almost entirely automatically — even for large real systems.

In our approach and tool, we reuse elements of ClawZ to reduce the development effort, and take advantage of validated tools and Z models. We directly incorporate the schemas generated by the Z Producer into our *Circus* model. In Fig. 1, the dotted line indicates the components that we added to the ClawZ toolset to cater for *Circus* models. An additional element of supplied information is the ClaSP block library. It contains essential information regarding the concurrent behaviour of primitive blocks. Like with the ClawZ library, we anticipate that for particular diagrams the ClaSP library may have to be extended. The mechanised *Circus* semantics [16] enables us to translate the *Circus* model into a ProofPower-Z encoding, and like in the ClawZ verification process we will use specialised high-level tactics to automate the refinement proof.

Circus adopts elements from sequential programming as well as process algebra. The fundamental constructs are channels, processes and actions [20, 7]. Channels are introduced through channel declarations and are required for com-

$$
\begin{array}{l}
\mathbf{process}\ \mathit{Debounce\_Process} \mathrel{\widehat=} (\\
\quad \mathit{Debounce\_\_LogicalOperator\_Process} \\
\qquad \{\!|\ \mathit{Flag}, \mathit{Debounce\_\_LogicalOperator\_out}, \mathit{end\_cycle}\ |\!\} \\
\qquad\quad \| \\
\quad \mathit{Debounce\_\_DataTypeConversion1\_Process} \\
\qquad \{\!|\ \mathit{Debounce\_\_LogicalOperator\_out}, \\
\qquad \mathit{Debounce\_\_DataTypeConversion1\_out}, \mathit{end\_cycle}\ |\!\} \\
\qquad\quad \| \\
\qquad \ldots \\
\quad \mathit{Debounce\_\_Terminator\_Process} \\
\qquad \{\!|\ \mathit{Debounce\_\_SzRFlipzFlop\_out2}, \mathit{end\_cycle}\ |\!\}) \setminus \\
\qquad\quad (\mathit{Debounce\_\_LogicalOperator\_out}, \\
\qquad\quad \mathit{Debounce\_\_DataTypeConversion1\_out}, \\
\qquad\quad \mathit{Debounce\_\_DataTypeConversion2\_out},\ \ldots\ )
\end{array}
$$

Fig. 2: *Circus* translation of the Debounce diagram presented in Fig. 3.

munication and synchronisation as in CSP. Processes can be defined either explicitly or in terms of process operators. An explicitly defined process is a sequence of paragraphs that specify its state, auxiliary actions, which use or change the state information, and a main action that defines the behaviour of the process.

Fig. 6 provides an example of an explicitly defined process and some prior channel declarations. In the process, we first introduce a state paragraph that declares two components, namely, $\mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay1\_state}$ and $\mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay2\_state}$; the set $\mathbb{U}$ represents a universal type in ProofPower-Z. Actions are defined by schemas, like in the case of $\mathit{Calculate\_Debounce\_\_Counter64Hz1}$, or by a mixture of sequential and CSP constructs, like in the case of $\mathit{Execute\_Time}$. CSP operators such as guarding, parallelism, interleaving and hiding can also be used. In $\mathit{Execute\_Time}$, for example, we moreover use variable declarations and assignments.

Circus also provides operators to combine processes. In the models of control law diagrams, we only require parallelism, channel renaming and hiding. Fig. 2 exemplifies the n-way alphabetised parallel operator where each process is associated with a set of channels on which it is required to synchronise. Renaming changes the names of channels within a process. Finally, hiding, also used in Fig. 2, internalises communication events over given channels.

To perform the translation from Simulink to *Circus* we further require a graph model of the diagram. It records the number of inputs and outputs, and independent flows of execution of each block. It additionally includes details of whether flows depend on enabling signals, or the order of arrival of their inputs.

The *Circus* model of a diagram defines channels to represent each of its inputs, outputs and internal wires, and a basic explicitly defined process for each block. In addition, the diagram itself is modelled by a parallel composition that combines the processes defining the blocks. For example, Fig. 2 sketches the translation of the Debounce diagram given in Fig. 3.

Each of the parallel processes results from the translation of one block in the diagram. The synchronisation sets include their interface, that is, input and output signals, as well as end_cycle. While the synchronisation on interface channels corresponds to the passing of signals between blocks, synchronisation on end_cycle ensures that a new cycle is commenced only after all blocks have finished their computation for the current cycle. The channels that correspond to internal wires are hidden to reflect the view of the diagram as a 'black box'.

The blocks are each translated to a centralised, explicitly defined process that lifts the functional Z specification produced by ClawZ; Fig. 6 gives an example: the translation of Counter64Hz1. We maintain the functional behaviour defined by ClawZ, but also accommodate the intrinsic parallelism. The rôle of the *Flow* action is to specify, through interleaving, the independent signal flows inside the block; in our example, though, there is only one flow. For blocks with state, the *Circus* process introduces a state paragraph. The purpose of the *StateUpdate* action is then to update the state; it is based on the ClawZ schema. In Fig. 6, for conciseness, we omit the ClawZ schema $Debounce\_\_Counter64Hz1$.

In the next section we explain the modifications and extensions to this strategy that have been suggested by its mechanisation.

3 Extended Translation Strategy

The experience we gained from implementing and applying our tool has given rise to three extensions to the translation strategy, which we discuss in the sequel.

Structure of Models

The translation strategy outlined above distinguishes the translation of the top-level diagram from that of its blocks. The diagram is represented by a parallel composition of block models, which are centralised, explicitly defined processes. This is appropriate if parallelism in the implementation is only at the level of procedures implementing block functionality.

If, on the other hand, subsystem blocks in the top-level diagram, or any other level of the diagram structure, are implemented by parallel procedures, representing them as parallel processes is more appropriate. This renders the architecture of the specification closely aligned to that of the implementation, and greatly reduces the effort in deriving and discharging the refinement conjecture.

Centralised *Circus* processes which are implemented by parallel procedures have to be decomposed during refinement. The strategy that can accomplish this is not as easily automated, as it requires the definition of coupling invariants relating the state of the centralised and the decomposed processes.

We propose that each subsystem block of the diagram may be *selectively* translated into either an explicitly defined, centralised process, or a parallel process. The decision can be governed by the architecture of a prospective implementation. The choices have an impact only on the automation of the

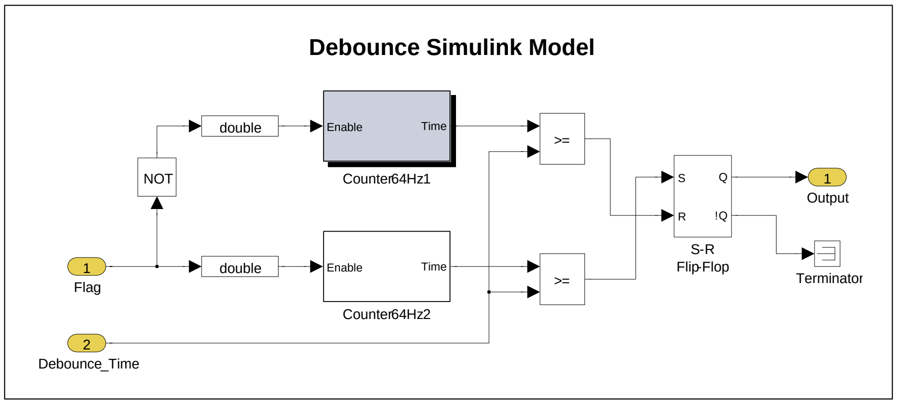

Fig. 3: Simulink diagram of the Debounce control law.

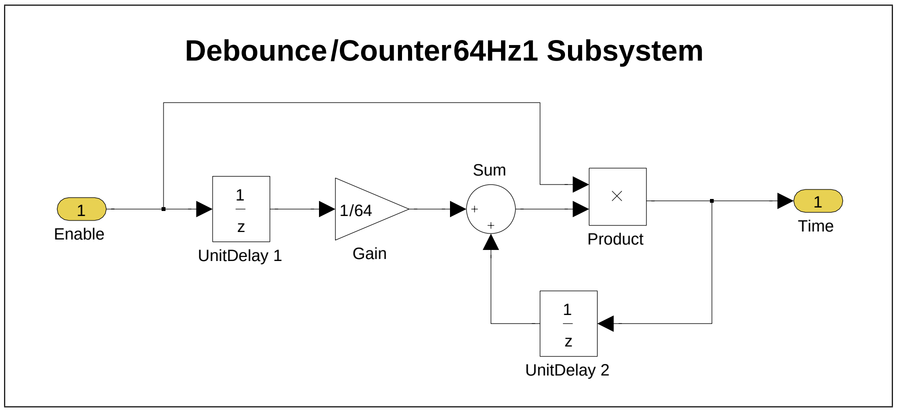

Fig. 4: Simulink diagram of the Counter64Hz1 subsystem.

verification; the models that are produced as the result of the different choices are semantically equivalent, and as a consequence they both capture the intrinsic parallelism in the diagram. This can be easily proved using laws of *Circus*.

To illustrate this point, we consider the Debounce control law given in Fig. 3. (This control system filters out a potential succession of quick oscillations upon toggling the state of a mechanical sensor or switch.) In the existing translation strategy, each element of the Debounce diagram would be translated into a centralised *Circus* process. Taking, for example, the Counter64Hz1 subsystem block, included in Fig. 4, the corresponding translation would read as shown in Fig. 6.

A possible implementation may choose to first compute the Time output signal, and then proceed by *concurrently* updating the state of the two UnitDelay blocks. Clearly, this parallelism is not directly reflected in the *StateUpdate* action of the centralised *Circus* model in Fig. 6. Using our extended translation strategy, we may decide to translate this subsystem in a parallel manner. It

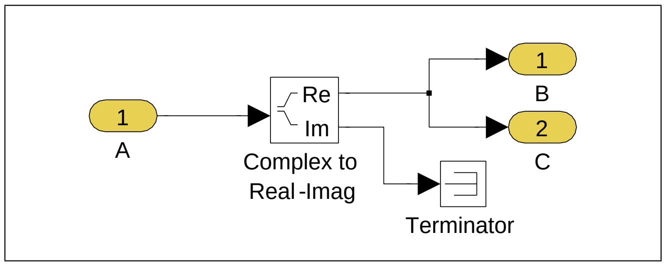

Fig. 5: Diagram causing problems for generating signal names.

adopts the same mode of translation we already used for the top-level diagram in the existing strategy (Fig. 2), but applies it to a subsystem. The process parallelism between the block translations exactly reflects our intention, for example, of implementing the UnitDelay1 and UnitDelay2 blocks by individual procedures.

Naming of Signals

Internal signals of a diagram are named according to the source block they connect. Clearly, there can only be one such block for each wire, although blocks can have multiple outputs. For blocks with only one output, the corresponding signal name is obtained by appending the suffix '`_out`'; for blocks with more than one output, the suffixes '`_out1`', '`_out2`', etc. are used.

An exception to the above rule are signals that connect input and output ports of the diagram. These are always named according to the respective port they connect in the model, i.e. `A`, `B` and `C` in Fig. 5. For input signals this proves not to be an issue, since there can only be one input port acting as the source.

For output signals, Fig. 5 depicts a scenario in which the signal name for the wire connecting Complex to Real-Imag, B and C cannot be uniquely derived as there exist two output ports potentially determining the name. To solve the problem, signal names are now always determined by their source location. In the example above, the name of the signal connecting the two output ports is `SignalNamingIssue__ComplextoRealzImag_out1`. We still, however, need to introduce signals for the outputs of the subsystem, through which it communicates with other blocks when instantiated in some diagram context. Hence there will be three channel declarations for our example.

$$\textbf{channel}\ \mathit{SignalNamingIssue\_\_ComplextoRealzImag\_out}1, B, C : \mathbb{U}$$

To communicate values to the output ports we take a view of them as *blocks* that simply pass on their input signal. This results in additional processes being created for each output port in the *Circus* translation, but the approach yields a very uniform treatment compatible with the fact that output ports are indeed represented as (Outport) blocks in the Simulink diagram.

$$
\begin{array}{l}
\textbf{channel}\ \mathit{Enable}, \mathit{Time} : \mathbb{U} \\[4pt]
\textbf{process}\ \mathit{Debounce\_\_Counter64Hz1\_Process} \mathrel{\widehat{=}} \textbf{begin} \\[4pt]
\textbf{state}\ \mathit{Debounce\_\_Counter64Hz1\_State} == \\
\quad [\mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay1\_state} : \mathbb{U}] \land \\
\quad [\mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay2\_state} : \mathbb{U}]
\end{array}
$$

$$
\begin{array}{|l}
\hline
\mathit{Init} \\
\mathit{Debounce\_\_Counter64Hz1\_State}' \\
\hline
(\exists\, b : \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay1} \bullet \\
\quad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay1\_state}' = b.\mathit{initial\_state}) \land \\
(\exists\, b : \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay2} \bullet \\
\quad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay2\_state}' = b.\mathit{initial\_state}) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\mathit{Calculate\_Debounce\_\_Counter64Hz1} \\
\mathit{In1}? : \mathbb{U} \\
\mathit{Out1}! : \mathbb{U} \\
\hline
(\exists\, b : \mathit{Debounce\_\_Counter64Hz1} \bullet \\
\quad \mathit{In1}? = b.\mathit{In1}? \land \mathit{Out1}! = b.\mathit{Out1}! \land \\
\quad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay1\_state} = b.\mathit{UnitDelay1.state} \land \\
\quad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay1\_state}' = b.\mathit{UnitDelay1.state}' \land \\
\quad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay2\_state} = b.\mathit{UnitDelay2.state} \land \\
\quad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay2\_state}' = b.\mathit{UnitDelay2.state}') \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\mathit{Calculate\_Time} == \\
\quad \mathit{Calculate\_Debounce\_\_Counter64Hz1} \setminus (\\
\qquad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay1\_state}', \\
\qquad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay2\_state}') \land \\
\quad \Xi \mathit{Debounce\_\_Counter64Hz1\_State} \\[4pt]
\mathit{Execute\_Time} \mathrel{\widehat{=}} \\
\quad \textbf{var}\ \mathit{In1} : \mathbb{U} \bullet \mathit{Enable}?x \rightarrow \mathit{In1} := x \mathbin{;} \\
\qquad \textbf{var}\ \mathit{Out1} : \mathbb{U} \bullet \mathit{Calculate\_Time} \mathbin{;} \mathit{Time}!\mathit{Out1} \rightarrow \mathit{Skip} \\[4pt]
\mathit{Flows} \mathrel{\widehat{=}} \mathit{Execute\_Time} \\[4pt]
\mathit{Calculate\_Debounce\_\_Counter64Hz1\_State} == \\
\quad \mathit{Calculate\_Debounce\_\_Counter64Hz1} \setminus (\mathit{Out1}!) \\[4pt]
\mathit{StateUpdate} \mathrel{\widehat{=}} \\
\quad \textbf{var}\ \mathit{In1} : \mathbb{U} \bullet \mathit{Enable}?x \rightarrow \mathit{In1} := x \mathbin{;} \\
\qquad \mathit{Calculate\_Debounce\_\_Counter64Hz1\_State} \\[4pt]
\bullet\ \mathit{Init} \mathbin{;} \\
\quad \mu\, X \bullet \mathit{Flows} \mathbin{[\![} \emptyset \mid \{\!|\ \mathit{Enable}\ |\!\} \mid \{ \\
\qquad\quad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay1\_state}, \\
\qquad\quad \mathit{Debounce\_\_Counter64Hz1\_\_UnitDelay2\_state}\} \mathbin{]\!]} \mathit{StateUpdate} \mathbin{;} \\
\qquad \mathit{end\_cycle} \rightarrow X \\
\textbf{end}
\end{array}
$$

Fig. 6: Centralised translation of the Counter64Hz1 subsystem.

Global Inclusion of the ClawZ Schemas

Finally, we avoid the inclusion of the ClawZ schemas in the local scope of the explicitly defined *Circus* processes. Initially this ensured that the Z schemas were only available in the scope in which they were used. It was an appropriate use of the modularity afforded by processes, but, for automatically generated models, it is not much of an advantage, and breaks the traceability between the *Circus* model and the ClawZ output. The actual inclusion of the ClawZ schemas takes place at a later stage when the *Circus* processes are semantically encoded into ProofPower-Z (see Fig. 1). Here, we directly incorporate the Z schemas from the respective ProofPower database generated by ClawZ upon model construction. This removes the need for reparsing the ClawZ-generated schemas when producing the *Circus* model and thereby erases the possibility of introducing errors in combining the Z and *Circus* models.

4 The *Circus* Producer

In this section we describe the main features of the *Circus* Producer — our tool for translating Simulink diagrams into *Circus*. We focus on the *usage* of the tool; its design and implementation are discussed in the next section.

User Interface The graphical interface of the *Circus* Producer is shown in Fig. 7. The structure of the given Simulink diagram is rendered as a tree where each internal node corresponds to a subsystem, and each leaf to a primitive block. The textual description for each node gives the name of the respective element in the diagram, and in parentheses its block type. The user may double-click on internal nodes to expand or collapse them. The functions **Expand All** and **Collapse All** expand or collapse all descending nodes of the current selection.

An important feature of the graphical interface is that *translation of the Simulink model is affected by the configuration of nodes in the tree control*: subsystem nodes that are expanded are translated in a parallel manner, and those collapsed are translated into centralised processes.

The context menu function **Show LaTeX** performs the translation of the (sub)diagram implied by the currently selected node; as a result, a LaTeX file is produced. The LaTeX directives used in the encoding are those of the *Circus* extension of the Community Z Tools (CZT).

By using LaTeX as the encoding language, viewable documents are easily generated for the produced processes. In fact, the **Show DVI** context menu function performs the translation as before, and additionally converts the generated LaTeX file into a DVI document and immediately displays it. The *Circus* process given in Fig. 6 was indeed automatically generated applying the *Circus* Producer to the subdiagram Counter64Hz1 of the Debounce model (Fig. 3, 4).

The ClaSP Library Part of the translation algorithm constructs a graph model of the diagram. Signal flows for subsystem blocks are calculated dynamically. For primitive blocks, however, we need to specify the number of input

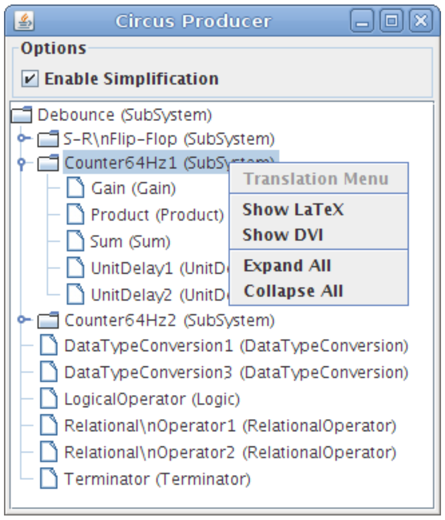

Fig. 7: Screen shot of the *Circus* Producer application.

and output ports, and signal flows. This information is included in the ClaSP block library.

To encode the library we use an XML-based format. The advantages of XML are, beyond standardisation and interchangeability, that the encoded file can be validated against the XML schema that defines its exact structure.

In general, the user only needs to make additions to the ClaSP library when blocks are encountered which the *Circus* Producer cannot translate. To help with their identification for a particular model, the *Circus* Producer generates warning messages each time a block is found that is not present in the ClaSP library.

To give an example of the format of library entries, Fig. 8 includes the ClaSP block specification of the UnitDelayWithPreviewResettable block. This block acts as a resettable unit delay with two outputs: one for the value of the input signal in the previous cycle, and an additional one for its current value. As the delayed output does not depend on the current input, the block has two flows.

The `<BlockType>`...`</BlockType>` pair of tags introduces the new type of block. The compulsory `name` attribute has to match the respective `BlockType` field in the Simulink encoding of the block. The optional boolean `state` attribute determines whether the block has state or not.

The aggregated `<BlockWiring>`...`</BlockWiring>` tags contain information about the inputs, outputs and flows of the block. The `<inps>`...`</inps>` and `<outs>`...`</outs>` tags specify the number of input and output ports. They can have a value `varlength`, if the block has a *variable* number of inputs, or outputs. In this case, the actual number of ports is inferred upon instantiation of the block in the model. The `<flows>`...`</flows>` tags include all independent signal flows of the block as an instance of `<flow>`...`</flow>`. In the example, there are two flows: one depends on both inputs, and the other on none.

```
<ClaSP>
  <BlockLibrary>
    ...
    <!-- Unit Delay with Preview Resettable (Additional Math & Discrete) -->
    <BlockType name="UnitDelayWithPreviewResettable" state="true">
      <BlockWiring>
        <inps>2</inps>
        <outs>2</outs>
        <flows>
          <flow>
            <enabled always="true"/>
            <ordered>false</ordered>
            <rinps>
              <portList>1 2</portList>
            </rinps>
            <pouts>
              <port>1</port>
            </pouts>
          </flow>
          <flow>
            <enabled always="true"/>
            <ordered>false</ordered>
            <rinps/>
            <pouts>
              <port>2</port>
            </pouts>
          </flow>
        </flows>
      </BlockWiring>
    </BlockType>
    ...
  </BlockLibrary>
</ClaSP>
```

Fig. 8: ClaSP Library extract for the UnitDelayWithPreviewResettable block.

The elements for each `<flow>`...`</flow>` instance correspond to our characterisation of the ClaSP model as formalised in [6]. Thus `<enabled>` specifies whether execution of the flow depends on enabling signals, `<order>`...`</order>` states whether the order of arrival of inputs is significant, `<rinps>`...`</rinps>` determine the set of input ports, and `<pouts>`...`</pouts>` the output ports of the flow. They can be given as individual ports (`<port>1</port>`), port lists (`<portList>1 2 3</portList>`), or ranges (`<portRange from="1" to="3">`).

***Circus* Model Simplification** Simplification is an optional feature of the translation which may be enabled or disabled by checking or unchecking an **Enable Simplification** check-box. This function only has an effect on the translation of primitive blocks which do not possess state. In such cases, the strict application of the translation procedure results in introducing vacuous state paragraphs, actions and schemas for performing the state update. Enabling simplification avoids the generation of these redundant parts of the process. The simplifications performed do not have an impact on the semantics of the translation; they merely aid readability and ease subsequent mechanical formal analysis.

5 Design and Implementation

We now discuss a few of the underlying design decisions and implementation issues encountered during the development of our tool.

Integration with CZT An important decision in the design of the *Circus* Producer was to integrate with CZT, the Community Z Tools, for the purpose of encoding and internally representing *Circus* specifications. CZT has been initially developed to provide a component library to facilitate development of Z tools [12]. Its open architecture, however, led to various extensions, including support for *Circus* [13, 9]. The integration with CZT, most importantly, avoids the need for a new design and implementation of a data model for *Circus* processes, since we can readily employ CZT's Annotated Syntax Trees (ASTs).

To process ASTs of *Circus* specifications, CZT's implementation of the Visitor design pattern endows us with a powerful and flexible mechanism to traverse and transform syntax trees. This will be especially useful for generating a semantic representation of *Circus* specifications in ProofPower, as required in further stages of our work. An additional benefit of integrating with the CZT component library is that we have the option to take advantage of existing or future CZT tools, such as the *Circus* type checker and model checker. This opens up opportunities for follow-up research using alternative approaches to reason about control laws in *Circus*, while exploiting the development effort we have already invested.

Architectural Considerations For the development of the *Circus* Producer application, we built a transparent library of well-designed, reusable data structures and components. This makes it easier to perform modifications and extensions to existing components, and, importantly, simplifies the development of new tools.

The tool has been structured into Java packages that deal separately with various aspects of data encapsulation, parsing, analysis and processing of data objects. The central data structures reside in the three sub-packages `Simulink`, `Diagram` and `ClaSP` of the enclosing package `Data` of the root application package. The classes in these packages are used to represent the Simulink file, the underlying Simulink diagram, and the ClaSP model, respectively.

The design objective was to make handling of the data objects as easy as possible for typical tasks, so that high-level functionality can be implemented with little effort. This is achieved by a tight linkage between data objects; for example, signals in flows are aware of the wire in the diagram they refer to, wires are aware of the blocks they connect, and so on. This allows information, such as signal names or the ClaSP wiring of blocks, to be computed dynamically, and reduces the data that has to be effectively managed by the application. To counteract a loss in efficiency resulting from dynamic computation, we integrated an annotation API similar to the one of CZT to cache information once computed.

Other packages provide tool components for parsing Simulink files, ClawZ library-meta files, generate Simulink diagram object representations from parsed files, and perform high-level operations on Simulink diagrams and, as we anticipate in subsequent versions, on *Circus* models. The modular architecture of these components allows for the quick prototyping and development of new tools; this, in particular, lead to the development of a collection of Java-based supplementary tools for ClawZ extending it and facilitating its use.

Representation of Blocks Each block in a diagram is represented by an object of a particular class. This means that developers of additional tools can introduce extra methods and fields specific to certain types of Simulink blocks, if needed. For example, the `Inport` and `Outport` classes representing the input and output ports of a diagram provide methods to obtain the port number of the blocks.

A potential complication is that upon extending the ClaSP library, classes must be created for each new block. This is a task that in practice should not be negotiated to the user. Therefore, such classes are generated automatically by a utility during compilation. The instantiation of block classes through a factory makes it possible for the developer to derive from the automatically generated classes in order to attach custom functionality if required.

Generation of the *Circus* Translation The various elements of the *Circus* processes resulting from the translation are described using the string template engine developed by Parr [17]. Templates are text files which contain placeholders to be 'filled in' when the template is instantiated. This isolates the static pattern of the translation from dynamic data that has to be provided to generate the concrete results such as names of processes, actions, signals, and so on.

The meta-language of the template engine defines constructs to address common cases such as generating lists within templates, conditional inclusion of text fragments, and the instantiation of one template from inside another.

We defined 22 templates to specify the translation rules. The use of string templates has so far proved very beneficial in terms of compacting the program code and facilitating changes and adjustments to the details of the translation, in particular since recompilation is not required when altering them.

6 Conclusions

In this paper we have extended the work in [6], where we describe a strategy to translate Simulink diagrams into *Circus* models, and presented tool support for mechanical translation. The extensions allow us to align the structure of the *Circus* specification with that of a given implementation, with the objective of simplifying the proof of refinement. The extended strategy also covers a wider range of wiring configurations, since it allows block outputs to be shared as diagram outputs. Finally, it simplifies the structure of the *Circus* model, and avoids the potential to introduce errors in the Z model which it aggregates.

Case Studies

The *Circus* Producer has been employed on a number of case studies of reasonable size which have served the purpose of validating the tool and evaluating its use. The examples have been provided to us by collaborators in the avionics industry, namely, EMBRAER and QinetiQ. The diagrams exhibit subsystem structure of up to 4 nesting levels, and the most complex of them contains a total of 155 elementary blocks and 14 subsystems making the construction of the *Circus* model by hand practically infeasible. Initially, applying the tool to these diagrams yielded only a partial translations due to incompleteness of the ClaSP library. At this stage, warning messages produced by the tool helped to identify the missing blocks, and subsequent consultation of the Simulink documentation to determine their behaviour in terms of flows.

The produced *Circus* models have been validated by inspection and comparison with the translation strategy in [6], and besides were tested for syntactic errors using the *Circus* parser of CZT. A further degree of validation will take place when future work semantically encodes these models in ProofPower-Z and applies the refinement strategy. We have compared the translation of the same diagram in different configurations, in particular to verify that parallelism is correctly represented in the sequential translation of subsystems. Notably, QinetiQ provided us with an example in which parallelism that surfaced at the top-most level of a digram revealed certain assumptions made about the environment by an implementation that were not explicit in the model.

As with ClawZ, the process of generating the *Circus* specification for control law diagrams can be largely automated, requiring a minimal amount of user interaction: the extension of the ClaSP library and the configuration of the translation of subsystem blocks as centralised or parallel processes. For both no knowledge of the underlying semantic details of the *Circus* representation is needed. This is very important as we would like most aspects of the verification to be driven by engineers without in-depth knowledge of the underlying verification strategy, let alone its formal justification. The *Circus* Producer, including its source code, is openly available for download from

`http://www.cs.york.ac.uk/circus/cld/tools.html`.

Related Work

In [3], an algorithm and tool is presented to translate Simulink diagrams into formal descriptions understood by the NuSVM model checker. Models are specified as finite state machines, hence the technique only applies to diagrams with finitary state spaces. The focus of this work is on automated verification of properties about the diagram; in comparison, we are concerned with proving conformity between diagrams and their implementations.

A formal semantics and tool support to reason about functional and timing aspects of Simulink diagrams is described in [8]. This work is based on the Timed Interval Calculus (TIC); tool support is provided for mechanical translation of TIC specifications generated from Simulink diagrams into corresponding specifications to be processed by the PVS theorem prover. This is again to validate properties of the diagram rather than to verify implementations. We currently do not characterise timing properties in our semantics; future work will consider the use of the timed extension of *Circus* for this purpose.

A Lustre model of Simulink diagrams is the object of the work in [4], which reports on the development of a strategy and translator utility. This can also be regarded as a formalisation of Simulink since Lustre, like *Circus*, is equipped with a formal semantics, including strong typing. The potential of this approach to produce implementations adhering to a high standard of reliability is, for example, in the use of certified code generators for specific target languages.

Future Work

At present, the automatic translation does not yet account for enabling signals which govern enabled, trigger or action subsystems. In [6] we already described how they should be handled, however work to extend ClawZ and the *Circus* Producer to accommodate these rules is still pending.

Another line of future work is the translation of StateFlow blocks, which permit the specification of subsystems using a notation based on State Charts. In [5], we described how *Circus* can be used to define models of StateFlow diagrams that can be used as components of a model of a Simulink diagram which includes them; automation is our next step.

The next phase of our work aims to translate the tool-generated *Circus* specifications into corresponding ProofPower-Z specifications using our semantic embedding of *Circus* proposed in [16]. We are then able to mechanically reason, within ProofPower, about the *Circus* specification, and apply our refinement strategy to show the validity of implementations. Automating the translation into ProofPower-Z shall only pose a problem of minor difficulty, however the mechanised proof of the refinement conjecture sets a substantial challenge.

Acknowledgements We benefitted from discussions with Phil Clayton, Leo Freitas and Daniel Bolton. We also acknowledge EPSRC for funding this work under the Programming from Control Laws research grant EP/E025366/1.

References

[1] M. Adams and P. Clayton. ClawZ: Cost-Effective Formal Verification of Control Systems. In *Formal Methods and Software Engineering*, volume 3785 of *LNCS*, pages 465–479. Springer, October 2005.

[2] R. Arthan, P. Caseley, C. O'Halloran, and A. Smith. ClawZ: Control laws in Z. In *3rd International Conference on Formal Engineering Methods*, pages 169–176. IEEE Computer Society Digital Library, September 2000.

[3] Meenakshi B., A. Bhatnagar, and S. Roy. Tool for Translating Simulink Models into Input Language of a Model Checker. In *Formal Methods in Software Engineering*, volume 4260 of *LNCS*, pages 606–620. Springer, November 2006.

[4] P. Capsi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating Discrete-Time Simulink to Lustre. In *Embedded Systems*, volume 2855 of *LNCS*, pages 84–99. Springer, September 2003.

[5] A. Cavalcanti. Stateflow Diagrams in *Circus*. In *SBMF 2008*, pages 1–16, 2008.

[6] A. Cavalcanti, P. Clayton, and C. O'Halloran. Control Law Diagrams in *Circus*. In *FM 2005*, volume 3582 of *LNCS*, pages 253–268. Springer, August 2005.

[7] A. Cavalcanti, A. Sampaio, and J. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2–3):146–181, November 2003.

[8] Chunqing Chen and Jin Song Dong. Applying Timed Interval Calculus to Simulink Diagrams. In *Formal Methods in Software Engineering*, volume 4260 of *LNCS*, pages 74–93. Springer, November 2006.

[9] L. Freitas, J. Woodcock, and A. Cavalcanti. An Architecture for *Circus* Tools. In *SBMF 2007: Brazilian Symposium on Formal Methods*, August 2007.

[10] B. Krogh. Approximating Hybrid System Dynamics for Analysis and Control. In *Hybrid Systems: Computation and Control, Second International Workshop*, volume 1569 of *LNCS*, page 2. Springer, March 1999.

[11] B. Krogh. Recent Developments in Modeling and Analysis of Hybrid Dynamic Systems. In *Application and Theory of Petri Nets 1999*, volume 1639 of *LNCS*, page 106. Springer, June 1999.

[12] P. Malik and M. Utting. CZT: A Framework for Z Tools. In *ZB 2005*, volume 3455 of *LNCS*, pages 65–84. Springer, April 2005.

[13] T. Miller, L. Freitas, P. Malik, and M. Utting. CZT Support for Z Extensions. In *Integrated Formal Methods*, volume 3771 of *LNCS*, pages 227–245. Springer, 2005.

[14] M. Oliveira, A. Cavalcanti, and J. Woodcock. Refining Industrial Scale Systems in *Circus*. In *Communicating Process Architectures*, volume 62 of *Concurrent Systems Engineering Series*, pages 281–309. IOS Press, September 2004.

[15] M. Oliveira, A. Cavalcanti, and J. Woodcock. Formal Development of Industrial-Scale Systems in *Circus*. *Innovations in Systems and Software Engineering*, 1(2):125–146, 2005.

[16] M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP semantics for *Circus*. *Formal Aspects of Computing*, Online First, 2007.

[17] Terence Parr. StringTemplate Engine. Available from http://www.stringtemplate.org.

[18] S. Ranville and P. E. Black. Automated Testing Requirements — Automotive Perspective. In *The Second International Workshop on Automated Program Analysis, Testing and Verification*, May 2001.

[19] The MathWorks, Inc. Simulink ®, 1994–2008.

[20] J. Woodcock and A. Cavalcanti. The Semantics of *Circus*. In *ZB 2002*, volume 2272 of *LNCS*, pages 184–203. Springer, January 2002.