Mechanised Translation of Control Law Diagrams into *Circus*

Frank Zeyda

HISE Group, University of York (UK)

10th of February 2009

Context

- Work funded by EPSRC project grant "Programming from Control Laws" (Ana Cavalcanti).
- Full-time researcher: Frank Zeyda, PhD student: Alvaro Miyazawa.
- Collaboration with QinetiQ (Colin O'Halloran, Phil Clayton, Nick Tudor) and Embraer (Alexandre Mota and Augustu Sampaio).
- Other collaborators: Marcel Oliveira whose PhD thesis provides a mechanisation of the *Circus* semantics in ProofPower-Z.

Project Aims

- Verification of Control Law Diagram (CLD) implementations using the *Circus* language.
- Building an infrastructure of tools to automate most aspects of the verification process.
- Benchmark: ClawZ suite of tools which was successfully used by QinetiQ to verify implementations of Simulink models of industrial-scale applications in the avionics sector.

Introduction and Motivations

- Engineers are comfortable with specifying control systems by means of control laws (Simulink is used as a *de facto* standard).
- Current research primarily focuses on verifying properties of the diagram; not much work is being done on the verification of implementations.
- For certification it is not sufficient to rely on code generators as the actual code is often constructed or optimised by hand.
  The verification of the code generator is a considerable challenge and cost factor in itself.
- Experience in industry shows that correctness proofs of control law implementations can effectively be automated.
- Verification of a larger class of diagrams and implementations.

Existing Solution

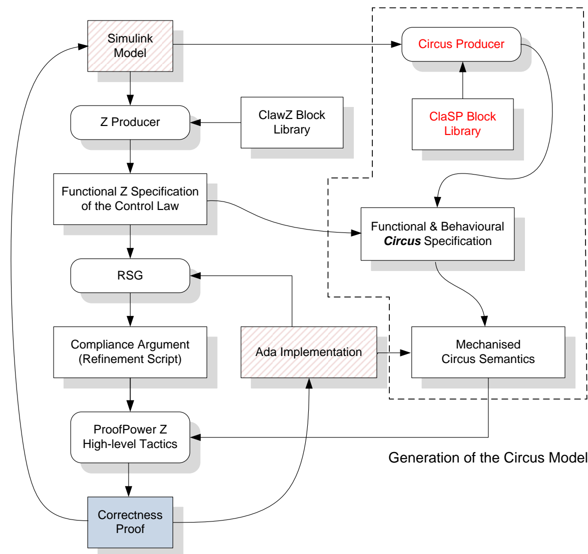
ClawZ Suite of Tools

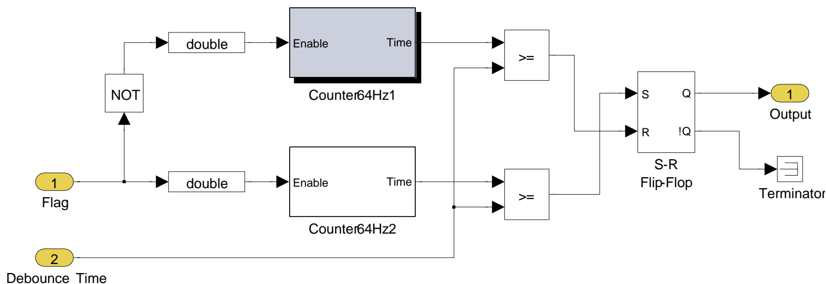
- The ClawZ suite of tools was developed by QinetiQ to verify (sequential) Ada implementations of Simulink models.
- ClawZ constructs a functional Z model of the diagram capturing its behaviour over one cycle of execution.
- Specifies the diagram by a set of schemas whose components include the inputs and outputs of blocks and subsystems.
- ClawZ can be used by engineers without in-depths knowledge of formal semantics and mechanical proof.
- Cost reduction of 20% was estimated in comparison to conventional development approaches of safety-critical systems in avionics.

Verification Proof

- Generation of a collection of refinement conjectures from the control diagram and code. Proof automated by high-level proof tactic.

ClawZ Tools Framework and *Circus* Producer





Example: Debounce Control Law

- Filters out a potential succession of quick oscillations upon toggling the state of a mechanical sensor or switch.
- Part of a larger case study provided by Embraer: controller for the elevator surfaces of a hypothetic aircraft. Click to open Z model.

Why use *Circus* to define a semantics for CLDs?

- ClawZ ignores the potential parallelism between the blocks.
- In principle, computations of blocks can be carried out in parallel, with order only imposed by their wiring.
- For example, in the Debounce model, Counter64Hz1 and Counter64Hz2 can compute their outputs independently.
- Even some basic blocks (UnitDelay) can produce their output before receiving all inputs. This can give rise to independent flows of execution within the block.
- The *Circus* model represents the parallelism between blocks in the diagram, and is closer to concurrent implementation.
- A refinement strategy can be used to collapse parallelism where necessary. This can be automated and easier than introducing it!

The *Circus* Language

Notable Features

- Integrated formalism that combines elements from both sequential programming and process algebra (Z and CSP).
- A *Circus* process encapsulates state as well as actions that operate on the state. The state is local to the process, and the behaviour of a process is defined by its main action.
- Actions may furthermore interact with the environment through communication events.
- Notation of *Circus* provides a mixture of CSP constructs, Z (operation) schema, and Dijkstra's guarded command language.
- *Circus* has an associated refinement calculus and mechanised denotational semantics based on the UTP.
- *Circus* is particularly suitable for the specification of state-rich, reactive systems. (State, for example, arises from Memory and UnitDelay blocks.)

Circus Model of a Diagram

- Blocks (or subsystems) are either defined as centralised processes or parallel compositions of processes.
- Transfer of signals between blocks is modelled by communications. For each wire we have a communication channel.
- Both representations reflect the inner parallelism of blocks, however one may be preferable to verify a given implementation.
- The functional behaviour is directly inferred from the ClawZ model, however lifted in such a way that flows of execution can be performed independently.
- Translation and refinement strategy has been published in "From Control Law Diagrams to Ada via *Circus*" (Cavalcanti, 2008).

- As a consequence outputs of blocks and subsystems maybe be calculated and produced before all inputs are provided.
- This aspects of the semantics of control laws is not captured in the functional Z model of ClawZ.

Example: *Circus* Model for Debounce

Translation of the top-level diagram.

$$
\begin{array}{l}
\textbf{process}\ \ Debounce_Process \mathrel{\widehat=} (\\
\quad Debounce__LogicalOperator_Process \\
\qquad \lbrace\!| Flag, Debounce__LogicalOperator_out, end_cycle |\!\rbrace \\
\qquad\quad \| \\
\quad Debounce__DataTypeConversion1_Process \\
\qquad \lbrace\!| Debounce__LogicalOperator_out, \\
\qquad Debounce__DataTypeConversion1_out, end_cycle |\!\rbrace \\
\qquad\quad \| \\
\quad Debounce__Terminator_Process \\
\qquad \lbrace\!| Debounce__SzRFlipzFlop_out2, end_cycle |\!\rbrace) \setminus \\
\qquad\quad (Debounce__LogicalOperator_out, \\
\qquad\quad Debounce__DataTypeConversion1_out, \\
\qquad\quad Debounce__DataTypeConversion2_out, \ \ldots\ )
\end{array}
$$

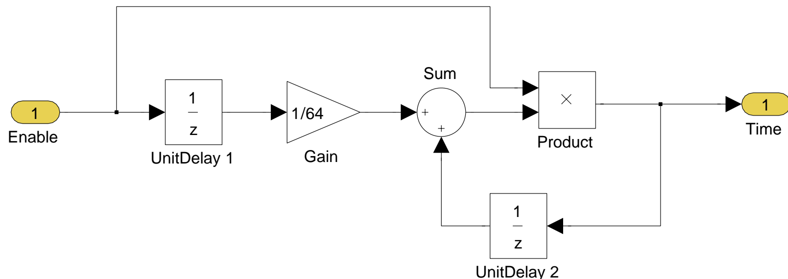
⇒ Represented as a parallel composition of the individual blocks.

Example: *Circus* Model for Debounce

Translation of the top-level diagram.

$$
\begin{array}{l}
\textbf{process}\ \mathit{Debounce_Process} \mathrel{\widehat=} (\\
\quad \mathit{Debounce__LogicalOperator_Process}\\
\qquad \{\!|\ \mathit{Flag}, \mathit{Debounce__LogicalOperator_out}, \mathit{end_cycle}\ |\!\}\\
\qquad\quad \|\\
\quad \mathit{Debounce__DataTypeConversion1_Process}\\
\qquad \{\!|\ \mathit{Debounce__LogicalOperator_out},\\
\qquad \mathit{Debounce__DataTypeConversion1_out}, \mathit{end_cycle}\ |\!\}\\
\qquad\quad \|\\
\quad \mathit{Debounce__Terminator_Process}\\
\qquad \{\!|\ \mathit{Debounce__SzRFlipzFlop_out2}, \mathit{end_cycle}\ |\!\}) \setminus\\
\qquad\quad (\mathit{Debounce__LogicalOperator_out},\\
\qquad\quad \mathit{Debounce__DataTypeConversion1_out},\\
\qquad\quad \mathit{Debounce__DataTypeConversion2_out}, \ \ldots\ )
\end{array}
$$

$\Rightarrow$ Represented as a parallel composition of the individual blocks.



Counter64Hz1 and Counter64Hz2 Subsystem

- Count the number of successive 1s divided by 64:

$$Time_n = Enable_n * (\frac{Enable_{n-1}}{64} + Time_{n-1})$$

- Unit delays blocks, for example, may be implemented by parallel procedures.

Example: *Circus* Model for Debounce

Translation of individual blocks as a centralised process (1).

channel $\mathit{Enable}, \mathit{Time} : \mathbb{U}$

process $\mathit{Debounce__Counter64Hz1_Process} \mathrel{\widehat=}$ **begin**

state $\mathit{Debounce__Counter64Hz1_State} ==$
$[\mathit{Debounce__Counter64Hz1__UnitDelay1_state} : \mathbb{U}] \land$
$[\mathit{Debounce__Counter64Hz1__UnitDelay2_state} : \mathbb{U}]$

Init

$\mathit{Debounce__Counter64Hz1_State}'$

$(\exists\, b : \mathit{Debounce__Counter64Hz1__UnitDelay1} \bullet$
$\mathit{Debounce__Counter64Hz1__UnitDelay1_state}' = b.\mathit{initial_state}) \land$
$(\exists\, b : \mathit{Debounce__Counter64Hz1__UnitDelay2} \bullet$
$\mathit{Debounce__Counter64Hz1__UnitDelay2_state}' = b.\mathit{initial_state})$

...

Example: *Circus* Model for Debounce

Translation of individual blocks as a centralised process (2).

$$
\begin{array}{l}
\textit{Calculate_Debounce__Counter64Hz1} \\
\quad In1? : \mathbb{U} \\
\quad Out1! : \mathbb{U} \\
\hline
\quad (\exists\, b : \textit{Debounce__Counter64Hz1} \bullet \\
\qquad In1? = b.In1? \land Out1! = b.Out1! \land \\
\qquad \textit{Debounce__Counter64Hz1__UnitDelay1_state} = b.\textit{UnitDelay1.state} \land \\
\qquad \textit{Debounce__Counter64Hz1__UnitDelay1_state}' = b.\textit{UnitDelay1.state}' \land \\
\qquad \textit{Debounce__Counter64Hz1__UnitDelay2_state} = b.\textit{UnitDelay2.state} \land \\
\qquad \textit{Debounce__Counter64Hz1__UnitDelay2_state}' = b.\textit{UnitDelay2.state}')
\end{array}
$$

$$
\begin{array}{l}
\textit{Calculate_Time} == \\
\quad \textit{Calculate_Debounce__Counter64Hz1} \setminus (\\
\qquad \textit{Debounce__Counter64Hz1__UnitDelay1_state}', \\
\qquad \textit{Debounce__Counter64Hz1__UnitDelay2_state}') \land \\
\quad \Xi \textit{Debounce__Counter64Hz1_State} \\
\quad \ldots
\end{array}
$$

Example: *Circus* Model for Debounce

Translation of individual blocks as a centralised process (3).

$$
\begin{array}{l}
Execute_Time \mathrel{\widehat=} \\
\quad \mathbf{var}\ In1 : \mathbb{U} \bullet Enable?x \longrightarrow In1 := x; \\
\quad\quad \mathbf{var}\ Out1 : \mathbb{U} \bullet Calculate_Time\ ;\ Time!Out1 \longrightarrow Skip \\[1ex]
Flows \mathrel{\widehat=} Execute_Time \\[1ex]
Calculate_Debounce__Counter64Hz1_State == \\
\quad Calculate_Debounce__Counter64Hz1 \setminus (Out1!) \\[1ex]
StateUpdate \mathrel{\widehat=} \\
\quad \mathbf{var}\ In1 : \mathbb{U} \bullet Enable?x \longrightarrow In1 := x; \\
\quad\quad Calculate_Debounce__Counter64Hz1_State \\[1ex]
\bullet\ Init; \\
\quad \mu X \bullet Flows \llbracket \varnothing \mid \{\!| Enable |\!\} \mid \{ \\
\quad\quad\quad Debounce__Counter64Hz1__UnitDelay1_state, \\
\quad\quad\quad Debounce__Counter64Hz1__UnitDelay2_state\} \rrbracket\ StateUpdate; \\
\quad\quad end_cycle \longrightarrow X \\
\mathbf{end}
\end{array}
$$

Generalisation of the Translation Strategy (1)

Structure of Models – 1

- Original translation strategy distinguishes the translation of of the top-level diagram from that of its blocks.
- Parallelism can, however, also surface at the level of subsystem implementations. Then it is more appropriate to represent the blocks as *parallel* process to avoid decomposition during refinement.
- We propose that subsystem blocks may be *selectively* translated in order to align it with a possible implementation.
- Decision is governed by the architecture of the implementation.
- The choice has only an effect on automation, but not the semantics of the diagram.
- Can be made without knowledge of the underlying reasons and impact on automation and proof.

Generalisation of the Translation Strategy (1)

Structure of Models – 2

$$
\begin{array}{l}
\textbf{process}\ \mathit{Debounce__Counter64Hz1_Process} \mathrel{\widehat=} (\\
\quad \ldots \\
\quad \mathit{Debounce__Counter64Hz1__UnitDelay1_Process} \\
\qquad \{\!|\ \mathit{Time}, \\
\qquad \mathit{Debounce__Counter64Hz1__UnitDelay1_out}, \\
\qquad \mathit{end_cycle}\ |\!\} \\
\qquad\quad \| \\
\quad \mathit{Debounce__Counter64Hz1__UnitDelay2_Process} \\
\qquad \{\!|\ \mathit{Enable}, \\
\qquad \mathit{Debounce__Counter64Hz1__UnitDelay2_out}, \\
\qquad \mathit{end_cycle}\ |\!\}) \setminus \\
\quad (\mathit{Debounce__Counter64Hz1__UnitDelay2_out}, \\
\quad \mathit{Debounce__Counter64Hz1__Gain_out}, \\
\quad \mathit{Debounce__Counter64Hz1__Sum_out}, \\
\quad \mathit{Debounce__Counter64Hz1__UnitDelay1_out})
\end{array}
$$

⇒ Similar to the translation of the top-level model we have seen before.

Generalisation of the Translation Strategy (1)

Structure of Models – 2

$$
\begin{array}{l}
\textbf{process}\ \mathit{Debounce__Counter64Hz1_Process} \mathrel{\widehat=} (\\
\quad \ldots\\
\quad \mathit{Debounce__Counter64Hz1__UnitDelay1_Process}\\
\qquad \{\!|\ \mathit{Time},\\
\qquad \mathit{Debounce__Counter64Hz1__UnitDelay1_out},\\
\qquad \mathit{end_cycle}\ |\!\}\\
\qquad\quad \|\\
\quad \mathit{Debounce__Counter64Hz1__UnitDelay2_Process}\\
\qquad \{\!|\ \mathit{Enable},\\
\qquad \mathit{Debounce__Counter64Hz1__UnitDelay2_out},\\
\qquad \mathit{end_cycle}\ |\!\}) \setminus\\
\quad (\mathit{Debounce__Counter64Hz1__UnitDelay2_out},\\
\quad \mathit{Debounce__Counter64Hz1__Gain_out},\\
\quad \mathit{Debounce__Counter64Hz1__Sum_out},\\
\quad \mathit{Debounce__Counter64Hz1__UnitDelay1_out})
\end{array}
$$

⇒ Similar to the translation of the top-level model we have seen before.

Generalisation of the Translation Strategy (2)

Naming of Signals – 1

- Internal signals are named according to the source block they connect. (There can only be one such block for each wire.)
- Blocks only one output have suffixes `_out`, and blocks with more than one output suffixed `_out1`, `_out2`, ... appended.
- Exception to this rule are signals that directly connect input and output ports. Their name is determined according to the port they connect.

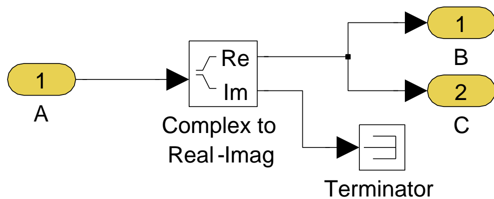
⇒ problematic diagram is given on the next slide!

Generalisation of the Translation Strategy (2)

Naming of Signals – 1

- Internal signals are named according to the source block they connect. (There can only be one such block for each wire.)
- Blocks only one output have suffixes `_out`, and blocks with more than one output suffixed `_out1`, `_out2`, ... appended.
- Exception to this rule are signals that directly connect input and output ports. Their name is determined according to the port they connect.

$\Rightarrow$ problematic diagram is given on the next slide!


1
A
Re
Im
Complex to
Real-Imag
1
B
2
C
Terminator

Generalisation of the Translation Strategy (2)

Naming of Signals – 2

- Solution to the problem: always determine signal names by their source location.
- Introduce outputs of subsystems as separate processes. They simply pass on the value from the internal wire to the output channel.
- In the example we have three channels:

  $\textbf{channel}\ \ \mathit{SignalNamingIssue__ComplextoRealzImag_out1}, B, C : \mathbb{U}$

  For each of the outputs B and C we have a separate (simple) process.
- Uniform treatment compatible with the fact that ports are indeed represented as (Outport) blocks in the Simulink diagram.

- Subtleties like these are often only discovered when actually implementing the tool support to automate the processes.

Generalisation of the Translation Strategy (2)

Naming of Signals – 2

- Solution to the problem: always determine signal names by their source location.
- Introduce outputs of subsystems as separate processes. They simply pass on the value from the internal wire to the output channel.
- In the example we have three channels:

  channel $\mathit{SignalNamingIssue__ComplextoRealzImag_out}1, B, C : \mathbb{U}$

  For each of the outputs B and C we have a separate (simple) process.
- Uniform treatment compatible with the fact that ports are indeed represented as (Outport) blocks in the Simulink diagram.

- Subtleties like these are often only discovered when actually implementing the tool support to automate the processes.

Generalisation of the Translation Strategy (3)

Global Inclusion of the ClawZ Schemas

- The schemas produced by ClawZ are part of the *Circus* model.
- Initially, the ClawZ schemas were explicitly included into the processes.
- This breaks traceability between the ClawZ and *Circus* model; the syntactic encoding of the ClawZ schema is not really necessary.
- We now include it directly from the ProofPower-Z database generated by ClawZ upon model construction.
- Less risk of introducing errors, for example when parsing and encoding the ClawZ schemas in *Circus*
- Ultimately we may consider supporting both solutions, but since our aim is to translate everything into ProofPower-Z, the current approach seems appropriate.

The *Circus* Producer – 1

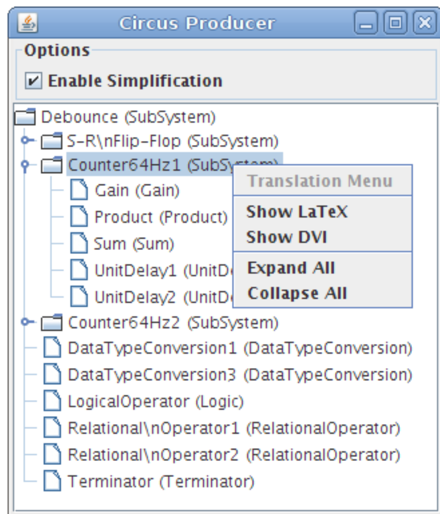
Main Features

- Automates the translation of Simulink diagrams into *Circus* models.
- Only information require for each individual subsystem is whether it should be translated in a centralised or parallel fashion.
- Graphical user interface.
- Encodes the *Circus* specification using LaTeX mark-up compatible with the Community Z Tools (CZT).
- Modular architecture of reusable components implemented in Java.
- Makes use of supplementary tools such as StringTemplate, XML parsing support for Java, and CZT.

Applications of the Core Library

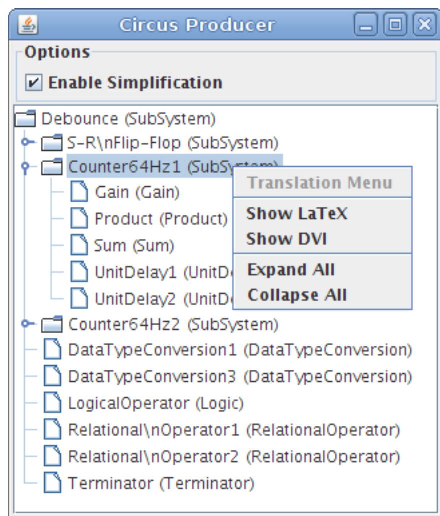
- Adoptions of Simulink MDL files $\geq$ version 4.0 to be processible handled by ClawZ.
- Analysis of models and student project in including type annotations.

The *Circus* Producer – User Interface



⇒ Demonstrate use of the tool translating the Embraer diagrams.

The *Circus* Producer – User Interface



$\Rightarrow$ Demonstrate use of the tool translating the Embraer diagrams.

The ClaSP Block Library

- Contains information regarding the flows of primitive blocks. (Flows of subsystems are computed dynamically using a graph model of the diagram.)
- Necessary for the translation to infer and propagate dependencies between the inputs and outputs of blocks.
- Encoded in an XML-based format.
- Can be extended by the user, namely to add support for custom blocks.

Example: Library entry for UnitDelayWithPreviewResettable

- This block has two outputs: one gives the previous value of the first input, and the second the current value.
- A second input is used to reset the current value.
- We observe that the block has two flows of execution.

The ClaSP Block Library (Example)

```
<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>
```

The *Circus* Producer – Design and Implementation

Architectural Considerations

- Aim to build a library of well-designed, reusable data structures and utility components.
- Structured into Java packages to separately deal with various aspects of data encapsulation, parsing, analysis, processing and model construction.
- Tight linkage between data objects to make processing as easy as possible and facilitate dynamic computation.
- Caching of data to increase efficiency, for example blockwiring information once the *Circus* model has been generated.
- Integration with CZT allows to reuse its data model and tool components for *Circus* models.
- Translation rules are encapsulated in templates using Parr's StringTemplate engine.

The *Circus* Producer – Design and Implementation

Example: String template for translating the *Flows* action.

```
\begin{circusaction}
  Flows \circdef\\\\
$if(flows)$
    $flows:{flow | \t1 Execute\_$flow.name$};
      separator=" \\interleave\\\\\n"$$nl()$
$else$
    \t1 Skip$nl()$
$endif$
\end{circusaction}
```

The *Circus* Producer – Design and Implementation

Representation of Blocks

- Each type of block in the diagram can (but need not) be represented by a particular class.
- Additional functionality (in form of methods) can be incorporated in the block classes.
- Base classes for blocks are constructed automatically from the ClaSP library.
- Instantiation managed through an object factory.
- Subsequent work plans to develop a type-checker for Simulink diagrams to extend the capabilities of ClawZ taking into account type information when constructing the Z model.
- Design will also facilitate subsequent extensions to support translation of StateFlow boxes (PhD project).

Case Studies

Embrear Case Study

- The *Circus* Producer has been validated by translating all diagrams of the Embraer case study.
- These diagrams contain subsystem structure of up to 4 nesting levels, and the most complex 14 subsystems and 155 elementary blocks.
- The case study has given rise to various extensions to the ClaSP block library.
- Size of the *Circus* model is far to complex for construction by hand.

NDI (QuinetiQ)

- Non dynamic inverstion controller. *Circus* translation and refinement was investigated as as part of a master project.
- Unveiled implicit assumptions made about the environment which had to be met for the given implementation to be correct.

Conclusion

- We extended previous work that described a strategy for translating Simulink control law diagrams to *Circus* specifications.
- The generalisation allows us to align the structure of the specification more closely to that of an implementation, easing refinement proofs.
- We presented a tool which completely automates this translation, and explained how its design fosters the development of further tools.

Future Work

- Automatic translation does not yet account for enabling signals which govern enabled, triggered and action subsystems.
- The library of blocks may subsequently be extended to cover custom Toolboxes of Simulink.
- Coneptual work has been done on translating StateFlow into *Circus*, and the translation shall be automaised as well.
- The *Circus* models will evenutally be tranlsated in a deep semantic encoding of *Circus* in ProofPower.