Lemma 1 Ltd.
c/o Interglossa
2nd Floor
31A Chain St.
Reading
Berks
RG1 2HX

ClawZ

—

The Semantics of Simulink Diagrams

Version:	7.4
Date:	12 May 2003
Reference:	DAZ/ZED503
Pages:	37

Prepared by:	R.B.Jones
Tel:	+44 1344 642507
E-Mail:	rbjones@rbjones.com

0 DOCUMENT CONTROL

0.1 Contents

0.2 Document Cross References

[1] LEMMA1/DAZ/ZED504. *ClawZ - Model Translator Specification.* R.B. Jones, Lemma 1 Ltd., `rbjones@rbjones.com`.

[2] *Using Simulink, Version 4.* The MathWorks Inc., 2001.

[3] *Target Compiler Reference Guide, Version 4.* The MathWorks Inc., 2001.

[4] *Using Matlab, Version 6.* The MathWorks Inc., 2001.

[5] *Writing S-Functions, Version 4.* The MathWorks Inc., 2001.

0.3 Changes History

Issue 2.1 First draft issue.

Issue 6.1 Cosmetic changes only.

Issue 6.2,6.3 Draft updates during the Real ClawZ design study.

Issue 7.2 Final update for Real ClawZ design study.

The scope of the document is extended to cover most of the things which are to be touched upon by the design study, particularly static analysis and target value spaces. The document has been substantially re-organised, the following list is an indication of the major areas of change.

- A clearer distinction is now made between *signal* values and *block parameter* values, which occupy different value spaces.
- Complex signals are now taken into account.
- A section on static analysis has been included.
- A section has been included discussing possible translator enhancements for the March 2001 updates to ClawZ.

Issue 7.4 May 2003 - Action Subsystems contract.

The main purpose of the changes in this version is to take account of developments to ClawZ which have taken place since it was last revised. The key relevant changes to ClawZ are the introduction of signal type inference, synthesis and virtualization capabilities for selected Simulink library blocks, dimension tracking while translating matlab expressions, and the implementation of support for Action Subsystems, which has general implications for the handling of non-transparent library blocks (such as the Action port).

0.4 Changes Forecast

No specific changes are forecast.

It now seems probable that ClawZ will continue to be developed by ad hoc increments tailored to specific additional requirements without any large strategic change. It will probably be more appropriate to document such requirements and their impact in separate documents rather than to update this document.

1 GENERAL

1.1 Scope

This document covers those issues in the semantics of Simulink models which either are, or may become, significant for the design, implementation or exploitation of the ClawZ Simulink model translator.

This version of this document is applicable to the version of ClawZ delivered in May 2003 and version 12 of Simulink.

1.2 Introduction

ClawZ is a tool that translates Simulink control law diagrams into Z specifications. The aim of the document is to discuss the key semantic issues which bear upon the design of ClawZ. This document is not intended to provide a thorough or systematic account of the semantics of Simulink models.

The main sources of information used in preparing this document were the Simulink tool and its HTML help files and PDF documentation, particularly:

1. The *Using Matlab* manual [4]
2. The *Using Simulink* manual [2]
3. The Simulink *Writing S-Functions* manual [5]
4. The Simulink *Target Language Compiler Reference Guide* [3]

This document was originally written to discuss the range of semantic issues that were investigated during the design of the first prototype of ClawZ. Since then, a number of enhancements to the design have been undertaken and this document has been updated to keep the information in this document correct relative to the current version of ClawZ. However, the formal statement of the current design is maintained in another document [1]. The information in this document that is specific to the current implementation could serve as the basis of a future user document giving an informal account of the operation of the ClawZ translator.

1.3 Notation and Conventions

The document makes no use of formal specifications other than fragments of Z in examples. A full specification of the current ClawZ translator is given in [1].

For lack of a better word I have used the word *operation* for the kind of mathematical object which is denoted by a Simulink block. This follows the usage in the Z literature, and consists of a relation which is interpreted as a transformation of some kind of state taking some input values and delivering output values (which will be signals).

Formalisation of some of the material is not excluded, but not anticipated.

1.4 Overview

The remainder of this document is organised into 9 sections discussing the topics summarised below:

Section 2 Preliminary Discussions

It is proposed that a clear distinction be made between the semantics of a diagram and the result of simulating the system described by the diagram. We discuss how the semantics can be deduced from the available descriptions of how simulation is done. Some of the issues arising from the infidelity of simulation are discussed, together with approaches to managing this problem.

This section introduces a factorisation of the semantics, reflected in separation of concerns within the ClawZ translator, and between the translator and the ClawZ library.

Section 3 Signals

The first and simplest stage in the discussion of semantics concerns signals. Signals are real or complex scalars or vectors.

Section 4 Blocks

Blocks, when fully instantiated may be considered to denote operations over signals. The representation of such operations in Z is discussed.

Transparent, opaque and virtual blocks are discussed here.

Section 5 Diagrams

An account of how blocks can be combined via connection diagrams to give new blocks. This section also covers virtualization, the absorption of blocks into wiring diagrams.

Section 6 Parameters

The semantics of blocks considers only the denotation of *fully instantiated* blocks, by which we mean a block together with the values of all its parameters.

An uninstantiated block (if this is a formal object) must therefore be some kind of function from parameter values to block values. This section is mainly concerned with the problem of instantiation.

Section 7 Expressions

The principle kinds of expression which may appear in Simulink block parameters are discussed.

Section 8 Static Analysis

Certain kinds of static analysis are identified and their possible roles in the semantic mapping considered.

Section 9 Translator Enhancements Options

Options for following up this improved account of the semantics of models by enhancements to the ClawZ model translator are identified and discussed.

2 PRELIMINARY DISCUSSIONS

2.1 Semantics and Simulation Results

Simulink uses numerical techniques to simulate the behaviour of continuous dynamic systems. These methods provide approximations to the behaviour of the systems rather than exact results. For this reason a variety of different simulation methods are offered, permitting the user to chose the method which gives best results for his application.

Among the sources of numerical error are:

1. Rounding errors arising from the use of floating point computation.
2. Errors arising from the use of finite rather than infinitesimal time periods in the computation of the integral.
3. Errors flowing from inaccuracy in locating discontinuities (associated with zero state values).

A purely numerical approach will suffice only where there are no zero-delay loops in the system. In the presence of such loops algebraic techniques are used to find a solution. These algebraic techniques may select a single solution where there are several, or may find none at all even though there may be a solution.

The simulator may therefore give different results from simulating the same model by different methods. Even with a single simulation method, the result may be sensitive to factors such as the order of evaluation of the blocks in the system, and hence to whether subsystems are atomic or virtual. A user must have an awareness of the possible inaccuracy of simulation results.

In the case of purely discrete systems the problems are less severe. All the simulation methods are alleged to give the same results on purely discrete systems. As far as I am aware the only source of infidelity in this case is the limitations in the ability of Simulink to solve algebraic loops (though there must surely still be the possibility of floating point rounding errors). It is doubtful whether algebraic loops with multiple solutions should be permitted in critical systems.

The distinction between the *meaning* of a diagram and its *behaviour under simulation* is sustained in this document and throughout the ClawZ specification and design. This does mean that it is possible that a system when simulated may give the desired results, and that a program generated by Simulink may give the same results, but that the program is not a correct implementation of the Z specification produced by the translator. We hope that this will not often be the case, and that this kind of disagreement between the semantics as interpreted by ClawZ and as implemented by Simulink can be eliminated by reasonable design guidelines (indeed, that it will not occur under established good practice).

In some ways a translation from specification to specification avoids problems which are encountered by a simulation. For example, where a specification is loose, a simulator must chose one interpretation, while a translation to another specification language can retain the looseness. The proposed translator will do just this with algebraic loops which have multiple solutions. The resulting Z specification will also be loose, and will have just the same number of models as the loop has solutions. If uniqueness is a required feature, then this may be demanded as a proof obligation.

Some of the detailed decisions in the design of ClawZ are simplified by this principle, which allows some aspects of the Simulink model to be ignored by ClawZ, among these are:

- Numeric datatypes are all interpreted by the single type of real numbers in ClawZ.
- No special measures are required by the translator to deal will algebraic loops.
- The distinction between atomic and virtual subsystems, which affects only evaluation order, is ignored.

2.2 Factorising the Semantics

In order to get maximum flexibility and power from a translator of modest complexity and cost, the semantics is factorised and responsibility for the various aspects of the mapping is divided between the hard-coded (and therefore relatively hard to change) translator, the ClawZ Z library and its metadata (which are simple enough to be extended or modified by ClawZ end users).

The primary basis for this separation is the distinction between the semantics of diagrams (which make new blocks from old, and are translated by the ClawZ tool) and that of the library blocks from which the diagrams are composed (which are hand translated into, or independently specified in Z, forming the ClawZ Z library).

In earlier versions of ClawZ this factorization was specified and implemented in a fairly clean way. As the tool has evolved enhancements such as support for parameter translation, block virtualization and synthesis have blurred this clean separation and embedded into the ClawZ translator more specific knowledge of the modelling of Simulink blocks.

Further discussion of the semantics can profitably be be broken into consideration of:

- signals
- diagrams
- blocks
- parameters
- expressions

3 SIGNALS

The generality of the relational modelling method that we will propose in section 4 below allows us to defer for the present, though not to omit mention of, the question of what kind of information is carried by the connections in a diagram. It is clear that the connections can carry scalar numeric values or vectors of these same numeric values. Most library blocks will handle signals with scalar or vector quantities, in many cases by performing the same scalar operation iterated or mapped over the vectors (these blocks are described as "vectorised" in [2]).

It remains uncertain whether signal values are best treated as a single untyped value domain, or handled using polymorphism or overloading. Though it seems doubtful that either polymorphism or overloading can provide a complete solution, they remain effective in providing cost effectively sufficient cover for typical applications. The use of a single value domain appears to offer an attractive simplicity. However, where the application requires only scalars, the use of this more complex domain of values would be likely to increase the complexity of compliance arguments.

With the proposed semantics for diagrams it is possible to experiment with different formal translations of the same libraries. It would then be possible to assess the relative merits of alternative modelling methods. This flexibility is in some measure reduced by the incorporation into clawz of the necessary specific information about signal types to undertake block virtualization and synthesis (though these features can be effectively disabled). In early versions of ClawZ these questions about the kinds of information carried by the connectors had little impact on the semantics of the diagrams or the implementation of the diagram translator, but the interaction is now more complex.

Matlab also permits other (non-numeric?) data types. It is not clear whether any of these are permitted (on "wires") in Simulink models, we have seen no evidence, as yet, that they are.

3.1 Data in Matlab and Simulink

According to the *Using Simulink* manual [2], Simulink supports all built-in Matlab data types (not user-defined types) and introduces the type *boolean*. The built-in types are:

1. double
2. single
3. uint8
4. int16
5. uint16
6. int32
7. uint32

Where *double* and *single* are floating point formats and *int* and *uint* are signed and unsigned integers of various widths.

Signals may be real or complex values represented by any of the above data types (presumably doubled up for complex numbers).

Booleans are a defined type based on uint8 values.

In Matlab these values can be combined into arrays of arbitrary dimensionality.

In many ways Simulink appears to behave as if only scalars and vectors are relevant for signal values, but the *Using Simulink* manual [2] (section 3.7) says that two dimensional arrays are allowed for some block types. The constraints (if any) on what kinds of arrays are permitted as parameters to blocks are not clear.

In Matlab an array has a single type, which is the type of its constituents, however this can be the type "cell", which effectively permits arbitrary constituents. In Simulink, though wide signals look like vectors, the signals passing through virtual blocks are effectively broken up by the diagram processing and so mixtures of values which would not otherwise be possible are permitted.

Typing appears to be primarily a run-time issue (the data typing rules are described as ensuring execution without error) though some checks are done as the model is created or loaded (e.g. consistent dimensions in matrices).

Signal and parameter datatypes are said to be independent, with some exceptions, e.g. the constant block. Where coercion is needed to match a signal and parameter it is usually the parameter which is coerced, not the signal.

3.2 Signal Values

Signals in Simulink have three characteristics:

1. They have a datatype selected from the list in the previous section.
2. They are either real or complex.
3. They have a positive integral width.

A signal of width 1 is called a scalar, one of length greater than one is called a vector.

3.3 Representation in Z

It is clear from the Simulink documentation, that the differences between the various numeric datatypes are only important from the point of view of simulation efficiency. The system defaults to the use of *double*, and the only effect of choosing a smaller datatype is that simulations may run faster, generated code may be smaller, or an out-of-range error may be provoked preventing simulation, compilation or code generation.

Semantically therefore, following from the discussion in section 2.1, it is reasonable to map all these datatypes to ProofPower-Z real numbers. Complex numbers could be mapped to pairs of reals and signals of width greater than one to sequences.

Since the exact width of a sequence has no bearing on its type, this would yield just four possible types for signals in Z, real or complex or sequences of real or complex values.

Since the complex numbers encompass the reals, and a scalar can be represented as a unit sequence (and there appears to be no need to differentiate between these so far as Simulink is concerned), either of both of these dimensions could be collapsed to give a smaller set of "types". A candidate for a "unified" signal type is the complex sequences in Z.

The designer of a library for use with ClawZ can chose which combination of these types he will use, the translator need not know since the types on the signals are not used in the block selection criteria used by the translator, and all the block selection criteria are made explicit in the metadata provided by the library designer. This freedom is however limited in practice by the hardcoding of signal representation choices in more recent ClawZ capabilities such as block virtualization and synthesis, and a user wanting to use a library with different signal types may find that he can no longer make use of these capabilities.

4 BLOCKS

In a denotational semantics for a complex language, the denotation of an expression may be expressed as a function from a type of data structures which represent the context in which expression is evaluated to a type to which the value of the expression belongs.

There may be considerable complexity in the kind of denotation required to capture the semantics of the expressions, even though the values computed are very simple. So, for example, in a simple language for arithmetic even though the values computed may all be integers, the denotation of an expression would be a much more complex value.

It is helpful in understanding the semantics of Simulink diagrams to focus upon what a Simulink block is when all these factors have been put aside, to get that idea of what a block denotes which corresponds to the idea that an arithmetic expression denotes an integer.

It is this kind of intuition about diagrams which is behind the example transcription undertaken by Alf Smith and which provides the fundamental basis for the proposed automatic translator.

The intuition is that discrete Simulink blocks are similar to the operations which are modelled in Z by a schemas. A block has a state, it has some inputs and some outputs. At each step in its history it computes from its input and current state an output and a new state.

In Z this function may be represented as an n-ary relation, which may in turn be represented as a set of labelled records. In the Z terminology such a set is called a *schema*, its elements are called *bindings*, the items in the bindings are called *components* and the labels are called *component names*. Apart from the terminology, this is closely analogous to the representation of a relation in relational database: a Z schema corresponds to a table in a relation database. In both cases, the idea is to represent as an n-ary relation what might otherwise have been thought of as a function or computation, and also, for ease in understanding, to use the component names (column names), to document the items the components (columns or fields) in each binding in a schema (row of a table).

In the Simulink User Guide a block is characterised by a combination of three functions.

1. a function which computes the output from the state and the input values
2. a function which computes the next value of the discrete components of the state
3. a function which yields the rate of change of the continuous components of the state

In the case of purely discrete systems this information corresponds to that available in an appropriate Z schema (though the relation represented by a schema need not be functional or total).

In the case of continuous systems, though this involves a significant extension to the normal usage of Z schemas, the necessary derivatives can just as well be represented in the relational form as can the new value of the discrete part of the state. The usage of Z schemas can, if desired, be extended to the continuous case.

Conventional usage of Z involves no temporal interpretation of Z schemas. In order for a Z schema to be used as a model for a Simulink block a temporal interpretation must be given. Since Simulink blocks can "feedforward" with zero delay the output value corresponding to any input/state combination is to be understood as a value which appears on the output ports of the block at the same time as the input values and state values from which it is computed. The new state appears at the next instant in time, which depends upon the sample time for the discrete component.

This "naive" interpretation of Simulink blocks as schemas provides the basis for an interpretation of diagrams as creating new blocks by wiring up already available blocks. i.e. for interpreting diagrams as ways of constructing new schemas by operations on already available schemas.

4.1 Transparent and Virtual Blocks

Two distinct classifications of blocks are used by Simulink and by ClawZ, both of these are relevant to discussion of the semantics.

Simulink describes blocks as *virtual* or *non-virtual*. A virtual block is one which can be effectively absorbed (by Simulink) into the wiring diagram. It appears as a block in the Simulink library, in the visual diagram of a system or subsystem, and in the .mdl file of a model which uses it, but it is eliminated during the Real Time Workshop build process and does not appear as a block in the .rtw file (though it may still be mentioned). That Simulink is able to eliminate the block does not however mean that ClawZ can do likewise, though in some cases it can.

ClawZ now has a capability for *virtualization* similar to that found in Simulink, though not for exactly the same block types. This permits the semantics of a certain blocks to be absorbed into the predicate part of a subsystem schema. In the case of typical *virtual* blocks such as *Mux* and *Demux* this amounts to the elimination of intermediate equations in the set of equations which represents the connections within the subsystem. Virtualization, as implemented by ClawZ can also be applied to some blocks which are not considered by Simulink to be virtual (see section 5.1.4).

For the design of ClawZ the terms *transparent* and *opaque* were introduced.

The description in Chapter 3 of *Using Simulink* [2] of the general characteristics of simulink blocks, and the slightly more detailed description in Chapter 1 of *Writing S-Functions* [5] form the basis for the semantic discussions in this document and the justification for the detailed translator specification in [1]. However, not all of the Simulink library blocks appear, even when fully instantiated, to have these "general characteristics". For this reason the term *transparent* was introduced to refer to those library blocks which when fully instantiated do have this general character. Blocks which do not have this character are called *opaque*, and present special problems for the model translator. Roughly explained, transparent blocks have a behaviour, they can be simulated, and their contribution to the behaviour of the system flows entirely from their own behaviour and the way they are connected in. Furthermore, this behaviour depends only upon the explicit input signals or parameters to the block and the state of the block. No other information is available. Opaque blocks may not themselves have a behaviour at all, and have an impact on the behaviour of the system which cannot be wholly explained by their behavioural interaction with the blocks they are connected to. If they have a behaviour, it may depend upon information not mentioned as available in the Simulink "general characteristics", e.g. on the size of the sample step.

The virtual blocks seem to include some of those which we describe as *opaque* together with some blocks, notably *Mux* and *Demux* which though technically transparent in our sense, are really just about wiring, and could therefore be treated as (complexes of) wires rather than as blocks.

We introduce the term *ambivalent* to designate transparent virtual blocks, which can be implemented as blocks with a rather simple behaviour or treated as part of the wiring. The term *irregular* is introduced for non-virtual opaque blocks.

Some blocks are virtual only under certain conditions.

The following table, adapted from Chapter 4 of *Using Simulink* [2] shows the virtual blocks, indicating the conditions for the block to be virtual and which blocks are ambivalent. We conjecture that most other blocks (i.e. most non-virtual blocks) are transparent. Opaque blocks can only be implemented by special action on diagram translation, and/or by some extension to the semantics of blocks. Ambivalent blocks could be implemented either by the diagram translator or in the ClawZ Z library.

Blockname	**Virtual?**	**Transparent?**
Bus Selector	always	yes
Data Store Memory	always	no
Demux	always	yes
Enable Port	always	no
From	always	no
Goto	always	no
Goto tag visibility	always	no
Ground	always	yes
Inport	unless directly connected to outport of conditional subsystem	no
Mux	always	yes
Outport	in a subsystem	no
Selector	except in matrix mode	yes
Subsystem	unless conditional or atomic	no
Terminator	always	yes
Test point	always	?
Trigger port	unless output port is present	no

In the decomposition of the semantics of Simulink models, the primary distinction is made between diagrams whose semantics is captured by the diagram translator, and the transparent library blocks whose semantics is captured in a library specification invoked by the specifications translating diagrams which include the relevant blocks. In this scheme the semantics of non-transparent virtual blocks is handled by the diagram translator. Their non-transparency prevents implementation through the library, and therefore requires that the semantics be provided by the diagram translator. The non-transparent virtual blocks for which support is available include inports, outports, action ports, and subsystems (which are always treated as atomic even if they are virtual in the Simulink model). Support for irregular blocks has not been considered (in fact there probably never will be any). None of these appears at present to be required for the QinetiQ applications of ClawZ.

Originally transparent blocks were supported only through the ClawZ Z library, but for selected block types virtualization and/or synthesis are now supported.

4.2 Transparent Uses of Library Blocks

Simulink library blocks should not be thought of as merely a collection of pre-prepared diagrams. They generally achieve effects which could not be realised by a user constructing diagrams of his own.

The special effects realised through incorporation of library blocks can be considered as falling into two major categories which we are here calling *transparent* and *opaque*. A library block is transparent if it denotes a component of the system whose behaviour can be relationally modelled and whose effect on the behaviour of the system is wholly accounted for by that relation. For a transparent library block our problem lies in instantiation, the process whereby the translator arrives at an expression in Z which denotes the required relation. A library block is opaque if its behaviour cannot be accounted for by locating a denoted relation and combining it with the other denoted relations in the standard manner.

Examples of opaque library blocks are the trigger and enable blocks.

Though the majority of instances of library blocks in diagrams can be interpreted in the proposed manner by Z schemas, the way in which the library blocks are instantiated is complex and stretches the Z type system beyond breaking point.

The following methods are used to resolve these difficulties:

- Polymorphic specifications are used when possible (i.e. in Z, generic specification)
- Parameterized specification are used, the actual parameters being translations into Z of some of the parameter values found on specific instances of the block in Simulink models.
- A kind of *overloading* is supported. This involves the provision of multiple Z specifications for a single Simulink block type, together with selection criteria based usually on the block parameter values which control the selection of one of these specifications to be instantiated for any particular occurrence of a block.

4.3 Discrete and Continuous Models

Another dimension of variability in Simulink models concerns whether the models are discrete or continuous.

We believe that whether a model is discrete, continuous, multi-rate discrete or hybrid, relational models can be used for which the relational join remains a correct account of the semantics of the composition of systems by diagrams.

The situation here is similar to that concerning the data types. A general modelling method could be used, which would cope with hybrid models, but this would be at the expense of additional complexity in reasoning about simpler models. The present proposal makes the core translator independent of this decision, and permits us to evaluate more than one approach.

For the record we note here some of the alternatives. It should be noted that these are conjectures about which modelling methods might be formally correct for *transparent* blocks, and that except for the single rate discrete case we have as yet no basis for judging whether the methods are tractable for non-trivial applications.

1. Single rate discrete.

   This simple kind of system can be modelling in a manner directly similar to the usual ways of modelling discrete systems in Z. The relational model of each discrete system, subsystem or block contains one component or column for each input line (conventionally decorated with "?"), one for each output line (conventionally decorated with "!"), one for each of an arbitrary number of state components, and a similar set (decorated with "′"s) which represent the values of the state components at the next moment in time. Each tuple in the relation shows a possible configuration of the input and output wires and the state at some instant in time, together with a set of state values which will supersede the current values at the next instant in time. The behaviour of a discrete system over time can be discovered by iterating the process of replacing the state variables with their new values in the context of successive values on the input lines. All the necessary information is in the tuples of the relationship.

2. Multi-rate discrete.

   Composition of discrete systems works properly, using the proposed method, provided that the composed sub-systems or blocks all use the same sample rate. If it is desired to combine discrete systems which have different sampling rates, one way of achieving this is by converting them all to the same rate. The lowest common multiple of the sample rates is a candidate. Conversion to a higher sampling rate involves including a small clock in the state which is used to inhibit changes other than at the original sample times.

3. Continuous.

   Since there is no next moment in time in a continuous model, the progression of the system cannot be represented by showing the next value of the state variables. Instead it is necessary to show the rate at which continuous state components are changing. A continuous system can therefore be modelled by a relation similar to that used for a discrete system except that the quantities involved must be continuous, and the primed components indicating how the state is changing take the value of the derivative of the relevant component. The behaviour of a continuous system over time is recovered from the relation by integration.

4. Hybrid.

   Few systems are purely continuous. A system with continuous and discrete components, where the discrete components all sample at the same rate, could be modelled by simply combining the single rate discrete and continuous modelling methods. In this case each state component has a component of the tuple which indicates its progression, but the interpretation of varies according to whether the component is discrete or continuous. In this case, in order to understand how the system behaves over time you need to have some information which is not explicit in the representation of the system. You need to know which components are discrete and which continuous, and you need to know the sample rate of the discrete components.

   Provided that the discrete sample rates are the same throughout (which could be forced by converting the discrete components to a common rate) this method of representation will compose correctly using the proposed semantics for diagrams.

5. Hybrid multi-rate

   The most general and the most complex representation which we mention is one involving explicit sample times for discrete components. In this representation each discrete state component has a pair of values which show its next change, a new value and a time interval. The new value for the discrete component of state is expected to take effect after the specified interval has elapsed. Since every discrete component of the state has its own interval/value pair the discrete components need not be clocked at the same rate (and they need not be clocked at a constant rate). This representation allows for composition of hybrid multi-rate systems without any need for conversion of sampling rates.

4.4 Multi-Port sequences

If library blocks such as “sum” and “mux” are to be supported without having to put a definition in the Z library for every arity at which it is used, then some way will be needed to map a Simulink library block with a variable number of ports onto a Z library block with a single bundled input, probably a sequence. This would either require the library instantiation to generate a schema definition for this particular instance which invokes the general library definition and is then used in the normal way by in the diagram, or else the effect could be achieved without introducing auxiliary definitions by elaboration to the diagram translation.

An alternative approach is now available through virtualization (section 5.1.4) or synthesis (section 4.5). The current version of ClawZ includes the capability to virtualize or synthesize “mux” blocks (as well as several other types), but not “sum” blocks.

4.5 Block Synthesis

Block synthesis is an alternative method of providing Z specifications for Simulink blocks. Synthesis involves the generation of such specifications automatically by ClawZ. It does not differ semantically from the effect which would be achieved by the use of the library.

Pragmatically it differs in the following ways:

1. The block synthesis capability is hard coded into ClawZ and cannot therefore be extended by users.
2. Synthesised schemas are used for only one instance of a block in a model, and the use of synthesis is therefore liable to increase the size of translated models.
3. Because synthesis uses hard coded block specific knowledge, it can be in some ways more powerful than the use of the library. For example, synthesis can deal with *Mux* blocks with any number of ports.

5 DIAGRAMS

The simple intuition behind the semantics of diagrams is that a diagram does two things. Firstly it combines a number of blocks together by some simple process of aggregation, i.e. the constituent blocks are present in an unaltered form in the resulting block (possibly replicated). Secondly, that the wiring of the diagram then constrains the behaviour of these blocks so that they work together to provide the behaviour of the resulting block. This constraint works by simple identification of the inputs of some blocks with the outputs of others, or with ports to the system or sub-system.

This effect is the same as that obtained in Z by a disjoint conjunction of the constituent schemas and then the addition of constraints which identify the values which are connected by the wiring diagram. The effect is similar to that of a join on the tables in a relational database. Rather than using schema conjunction the ClawZ translator currently constructs a schema of which the components are bindings, one for each block in the system or subsystem diagram. This is logically equivalent but simplifies the problems which arise in preventing accidental identification of distinct blocks which have the same name in different diagrams.

It is a strength of this informal account of the semantics of diagrams that it is applicable independently of most of the fine detail in the semantics of Simulink blocks and diagrams, and therefore provides a basis for the core of a translator which will be stable while the broader semantic picture evolves to encompass more aspects of the detailed semantics of Simulink library blocks.

This model is general because it addresses just these two aspects, aggregation and wiring. The internal structure of the blocks which are combined in this way involves more semantic complexity, since these blocks can be parameterised in quite exotic ways, but these ways of constructing blocks do not impact the semantics of wiring diagrams which assumes that the blocks on which it operates are fully instantiated.

5.1 Virtual and Virtualizable Blocks

Though we believe that the key semantic feature of diagrams is captured by the relational join operation, there are some features which do not appear to fit. These exceptions are generally associated with the use of what we are here calling *opaque* library blocks.

Some transparent blocks, (the ones we describe as *ambivalent*) can either be interpreted as blocks or as a shorthand for wiring. In addition, transparent blocks which are purely functional (i.e. have no internal state), whether or not they are considered by Simulink to be virtual, can be absorbed into the wiring equations by using an appropriate expression rather than a simple equation (this process is known in ClawZ as *virtualization*).

The virtual blocks, consisting the ambivalent blocks and some opaque blocks, and other blocks which are virtualizable because they have no state, are therefore relevant to the semantics of diagrams and are discussed below.

5.1.1 Input and Output Ports

Technically, the input and output ports represent an opaque use of library blocks, since they are not considered to denote a relation which is composed with the relations denoted by the other blocks. However, the role of these blocks in identifying the external ports is fully taken account of by the diagram translator in the way in which the denotations of the other blocks are composed to form the overall system.

5.1.2 Mux and Demux

These blocks are ambivalent and in early prototypes of the translator have been implemented in the same way as other transparent library blocks. The fact that they have variable numbers of ports, and that the width of the signals to these ports may vary arbitrarily (subject to conservation of width across the block as a whole), makes their support awkward without support for variable numbers of ports.

These blocks can be absorbed into the diagram of the system or subsystem in which they occur. This results in simplification of the Z schema, including a reduction in the size of its signature. This feature is now supported by ClawZ.

5.1.3 Action, Trigger and Enable Ports

These ports represent a special kind of block which when included in a subsystem makes the execution of the subsystem conditional and provides a way of controlling its execution.

If a trigger port block is included in a diagram then a new input port is created and the subsystem defined by the diagram will undergo a transition only when an edge is presented at this port.

If an enable block is included then a new input port is created and the system defined by the diagram will undergo state transitions only under the control of the signal presented at this port.

Inclusion of an action port in a subsystem permits the subsystem to be controlled by an *If* or a *SwitchCase* block.

These port block types are not transparent because they effect the behaviour of the entire subsystem of which they are a part in a way which cannot be achieved by simple composition of the behaviour of the port block with that of the other components.

The correct semantics for the conditional subsystems which contain these ports can be realised by the use of multiple Z schemas for the subsystem. One schema represents the normal operational behaviour of the subsystem, and is the same as the schema for the subsystem would have been had the special port been omitted, apart from one extra entry in the signature of the schema for that port. Other schemas represent the behaviour of the subsystem when its normal action has been suspended. This usually involves either preservation or reset of the state and output values, depending on the parameters on the special port and/or the subsystem output ports. The composite action of the conditional subsystem can then be realised in Z using a schema expression in which the value on the special port selects the appropriate behaviour of the subsystem.

In the case of Action subsystems special measures have to be taken to get the semantics of the Merge block right. This is because the subdiagram containing an *If* or *SwitchCase* block, the subsystems which it controls and the *Merge* blocks which collate output from these subsystems is not technically compositional. The *Merge* block is required to transmit to its output the input which it receives from the currently active action subsystem, even though it appears to have no way of telling which subsystem is active.

Two possible solutions to this problem are:

- integrate the value on the action port into the outputs from an action subsystem

- provide separate connections from the outputs from *If* or *SwitchCase* blocks to the relevant *Merge* block.

The second of these has been adopted in ClawZ.

5.1.4 Virtualization

Virtualization consists of incorporation into the wiring equations of a subsystem schema the functionality of some of the blocks which occur in the subsystem. It is most obviously applicable to transparent virtual blocks such as *Mux*, *Demux*, *BusSelector* and *BusCreator*, but can also be done for any transparent block which has no internal state, e.g. *Constant* blocks.

The effects on the subsystem schema are as follows:

1. The names of the virtualized blocks do not appear in the signature of the subsystem schema.
2. For each output of a virtualized block an expression is computed, and is then used in those places in the predicate of the subsystem schema where the name of the relevant output port would otherwise have been used.
3. In the simplest cases the expressions used will be expressions in the names of the output ports on blocks in the subsystem which are connected to the input ports of the block being virtualized, but if these are output ports on blocks which are themselves being virtualized then the relevant expression must be used instead of the port name (which will not be available because the block will not appear in the signature of the system or subsystem schema).

5.1.5 Exotic Connections

The present proposal supports only connections to input, output and action ports, to sources which are independent, or to sinks which exert no further influence on the future of the system.

Connections to enable and trigger ports are partially supported, mainly in connection with stateflow systems. Stateflow subsystems are accepted and a stub can be generated, but no semantics is given to such subsystems.

No consideration has so far been given to the feasibility of supporting other kinds of connection.

5.2 Masked Subsystems

Simulink subsystems may be parameterized and are then called masked subsystems. Masked subsystems may be included in a library and reused in multiple models.

The following steps are taken by the ClawZ translator to support masked subsystems:

- The translator keeps track of what mask variables are introduced and where they are used.
- Any subsystem in which a mask variable is used is translated as a lambda abstraction in which the argument is a single binding with one component for each mask variable used in the subsystem (not necessarily at the top level). The body of the abstraction is a horizontal schema similar to the schema which would have been used had the subsystem not contained occurrences of mask variables.
- When the subsystem schema is used in the declaration part of its enclosing subsystem, the translated actual mask parameters to that subsystem and also the variable names for any mask variables which are used in that subsystem though not declared by it as mask variables, are supplied in a binding as an actual parameter to the function defined by the lambda abstraction.
- When reference is made to a library block in whose parameter values there are occurrences of mask variables, the mask variables must be passed to the instance of the relevant specification declared for this particular occurrence of the block type (which must therefore be defined as an abstraction when library instantiation takes place). An appropriate binding is used.
- If the subsystem is an Action subsystem, then multiple schemas will be defined. Each of the schemas is defined as a lambda abstraction and is supplied with a suitable parameter when used. Similar action is necessary for any *state held* or *state reset* schemas corresponding to a subsystem schema which must be parameterised under this scheme. These schemas must be abstractions when declared, with the same signature as the subsystem schema to which they correspond, whether or not they actually use the same mask variables.

6 PARAMETERS

A wide variety of effects can be achieved by configuring the parameters of Simulink library blocks.

To get reasonable coverage in ClawZ for translating Simulink library blocks by reference to specifications in the ClawZ Z library it is necessary to exploit effectively generic specifications, overloading and parameterization of specifications.

6.1 Uses of Library Block Parameters

Before considering how to achieve the required semantic effects we mention some of the effects which can be realised through parameters.

Simple Numeric Parameters These are ubiquitous, used for such basic purposes as specifying the value of a constant source or the slope of a gain block.

Vector Parameters Most places where you can put a scalar you can also put a vector. This is made more awkward by the fact that you may not be able to tell from the block parameters whether the ports are connected to scalars or vectors.

Optional Ports Many components have one or more ports which appear only if a checkbox is selected on the parameter form.

Variable Port Numbers Some components have a variable number of ports for some purpose with a numeric parameter which sets the number of ports.

Expressions Often expressions which determine or influence the functionality of a block are entered as parameters. In many cases these will be scalar constants or a vector of constant coefficients, but parameter values can be supplied as arbitrary matlab expressions (e.g. for a constant block) where the values associated with variable names are set by .m files or subsystem masks. Special expression languages (not the same as matlab expression) are defined for Fcn blocks, and special ways of passing values into such expressions are also used (e.g. in *If* block parameters).

Exotica There are probably a lot of these only to be discovered by a thorough sweep of the available libraries. A small example is the "sum" block, which takes a variable number of input ports and has a parameter which is a string of "+" and "-" signs determining for each input respectively whether it is added or subtracted from the total.

6.2 Approaches to Implementing Parameterisation

6.2.1 Overloading

The basic mechanism here is to use the values of certain block parameters to select an appropriate specification from the ClawZ Z library. The principle selection criterion is of course the block type, but any other block parameter can be used for selection, and there may therefore be multiple Z specifications used to give the semantics of a single Simulink block type.

For example, where a library block comes in several varieties with different combinations of ports, the port configuration can be used in the selection criteria for the Z SCHEMA which will be used to model the block. This is achieved by including in the library metadata information about parameter values which is used as a selection criteria for instances of blocks which will be mapped to a schema.

Selection is also effected by the translation of the values of block parameters into Z, failure of which will inhibit selection. The metadata for a Z schema in the library will indicate which parameters must be translated for transmission to the block, and will also give a parameter translation code which determines what kind of parameter is expected and how it is to be translated. If the supplied parameter is not suitable for the specified method of translation then the selection of the relevant Z specification is inhibited.

6.2.2 Parameter Passing

Some kinds of parameters are be supported by mapping to parameter passing mechanisms available in Z. The main limitations on this kind of solution arise from the type discipline imposed by Z, which makes the use of overloading essential for some of the effects achieved in the Simulink library.

The parameterisation of library blocks is supported by the use of functions which require a set of parameters gathered together into a binding and return a Z schema which is the relation denoted by the instantiated block. e.g. for a scaling block in which the scaling value was a parameter to the block the type of the block before instantiation would be something like:

z

$$\mathit{scale}{:}[\mathit{gain}{:}\mathbb{Z}] \;\rightarrow\; \mathbb{P}[\mathit{in}?,\ \mathit{out}!{:}\mathbb{Z}]$$

The specific parameter value supplied on instantiation of the Simulink library block would be translated in Z into a binding construction which is supplied as an argument to the scale function when it is used in the Z translation of the system diagram.

Even with support for some such parameter passing mechanism, there are difficulties arising from the kinds of parameter which are supplied to Simulink library blocks. The parameters need not only to be passed, but they must first be translated into Z before they can be passed.

6.3 State held and State reset Schemas

To provide support for conditional subsystems it is necessary to supply, alongside every Z specification of the normal behaviour of a Simulink block, specifications for the behaviour of the block when its normal behaviour is inhibited. Two cases are required, one when the state is to be reset, and one where it is to be held. These cases only cover Action subsystems where output values are to be held, and more elaborate provision along the same lines would be necessary to realise fuller support for conditional subsystems. These additional schemas must have exactly the same type as the main schema and will be instantiated identically.

6.4 Instantiation

In the Z specification which translates a model each instance of a block is given a named instantiation. i.e. a new schema is defined whose name is particular to that instance and whose value is the value obtained by supplying any relevant parameter translations to the function defined in the Z library. In the case that no parameters are required, the name is defined as a synonym of the specification in the library.

In the subsystem schema for the subsystem which contains the block instance the name of the instance is used in the declaration. If the translations of the parameters include the names of masked variables these would be out-of-scope at the point of use (i.e. in the definition of the instance) unless passed as parameters. In this case the instance is therefore defined as a function, and is supplied the mask variables as an argument when used in the declaration part of the relevant subsystem schema. The mask variables are passed as a binding, which contains just the mask variables which are used in the translations of the actual parameters to the library schema.

Using the *scale* example again, if a scale block in a model was given an expression $a + b$ as a gain parameter, where a and b were the names of mask variables, then the instance of the ClawZ Z library specification, defined for that block in the model, would have a specification along the following lines:

z

$$(path\_scale{:}[a{:}U;\ b{:}U]\ \rightarrow\ \mathbb{P}[in?,\ out!{:}\mathbb{Z}])$$
$$==\ \lambda\ pars{:}\ [a{:}U;\ b{:}U] \bullet\ scale\ (gain\ \widehat{=}\ pars.a\ +_R\ pars.b)$$

When this instance is invoked in the declaration of the subsystem containing the instance of the scale block it would appear as:

z

$$scale\ :\ path\_scale\ (a\ \widehat{=}\ a,\ b\ \widehat{=}\ b)$$

6.5 Masked Subsystems and Block References

A masked subsystem is translated into a horizontal schema definition in a lambda abstraction which abstracts over not only the mask variables but over any mask variables declared in higher subsystems and used in the masked subsystem. A subsystem even if not masked will be translated into a similar abstraction if it makes use of variables masked in higher level subsystems.

Subsystems, can be invoked by block reference, and these references can be translated provided that the metadata contains an appropriate entry for that subsystem. In that case the instantiation must supply values both for the masked variables of the subsystem and for any other mask variables used in the subsystem. The actual parameter values supplied for the mask variables are then taken and translated from the parameters of the block reference. The actual parameter values supplied for the mask variables used but not masked by the subsystem will simply be the name of the variable.

ClawZ is able to compile a library, in which case it not only produces a Z translation of the blocks and subsystems in the library, but also produces metadata suitable for block references to the subsystems in the library. Translation of block references is likely to work only for references to subsystems of libraries translated by ClawZ. This is because ClawZ can take the mask specification as a guide to how the translated specification should be parameterized, but has no comparable information available for other blocks. Block reference to blocks other than subsystems would be possible if the metadata and the Z specification for the block referred to were both written by hand.

7 EXPRESSIONS

Expressions may appear in a Simulink model as parameters to the blocks from which the model is built. A translator capable of translating arbitrary Simulink models into Z must be able to translate these expressions.

Four distinct kinds of expression are discussed here:

- Fcn block parameters
- other block parameters

- expressions in .m files
- *If* block parameter expressions.

7.1 Fcn Block Parameters

The Fcn block computes a function over a vector input signal delivering a scalar output. The function computed is determined by an expression supplied as a parameter to the Fcn block in a "C-like" language defined in the Simulink documentation for the Fcn block. The value of the input vector may be accessed in the expression as the vector u, which may be subscripted using either round or square brackets (though the admissibility of square brackets does not seem to be mentioned, they are used in the example in the *Using Simulink* manual [2]). u without subscript is interpreted as the first element, and no array operations (apart from subscripting) are permitted in the expression. The ClawZ implementation does not permit "u" to be used both with and without subscripts in the same expression.

7.2 Matlab Expressions as Parameters

In general expressions may be used in block parameters, though unlike the specific case of the Fcn block, block documentation does not mention what kind of expression is permitted.

I have found no description of what expressions are allowed. However, when an expression is rejected (which is not usually until an attempt is made to run a simulation) a Matlab error message appears, suggesting that the expression has been passed to Matlab for evaluation and that the expressions are therefore likely to be Matlab expressions.

In this more general case the expression is evaluated in the context of the Matlab workspace, possibly augmented by the values assigned to the parameter variables of masked subsystem blocks, and the variable u has no special significance.

7.3 Expressions in .m Files

Expressions in .m files are of course Matlab expressions. There does not appear to be any concise description of the expression syntax, which however is simple enough that reasonable subsets suitable for translation by ClawZ can be specified partly by consulting the available documentation and partly by checking against the behaviour of Matlab.

7.4 If Block Parameter Expressions

There are two parameter types for *If* blocks. The first supplies a single expression for the *if* condition, the second supplies a comma separated list of expressions for *elseif* conditions.

The expressions permitted are very simple truth functional combinations of elementary comparisons in which constants and input port values are permitted as operands yielding a boolean result. The input port values are entered in the expression using a variable name "un" where n is the input port number. The semantics is therefore very similar to the matlab and Fcn expression semantics differing primarily as necessary for the mechanism for incorporating input port values. The expressions could be translated into lambda expressions in Z whose type depends upon the number of input ports.

These are not yet supported in ClawZ.

8 STATIC ANALYSIS

Static analysis is here intended to cover all kinds of analysis of a model falling short of or prior to simulation or full formal reasoning. The boundary between static and dynamic is somewhat arbitrary, but nevertheless pragmatically significant. This definition includes syntactic analysis.

8.1 The Representation of Types

There are multiple type systems of relevance here:

1. the types of matlab expressions
2. the types of signals in simulink models
3. the types of Fcn block parameter expressions
4. the types of block specifications in the ClawZ Z library
5. the types of Matlab operators and functions in Z
6. the types of Fcn block operators and functions in Z

8.1.1 Matlab Expression Types

I have found no clear and concise description of the matlab type system. The closest I have found is the section on Data Types in the chapter on M-file programming in the *Using Matlab* manual [4].

The Matlab data values denoted by expressions are multidimensional arrays of values, having minimum dimensionality of 0 by 0 and no maximum number of dimensions. Scalars are 1 by 1 arrays. All arrays are effectively stored as one dimensional arrays, and can be accessed in this way. All indexes start at 1, the first (row) being the fastest changing then column then page. Thus: A(1,1,1)= A(1) and A(r,c,p) = A(r + (c-1)*rows + (p-1)*rows*columns). These values may be of type char, double, int8, uint8, int16, uint16, int32, uint32, struct or cell.

Of the numeric types only double is used by matlab for computation, the others are used for efficiency in storage or computation in generated code. Structures may be user defined, and form the basis for an object-oriented programming paradigm. Strings are one dimensional arrays of characters. Cells are themselves multidimensional arrays, and therefore permit arrays of strings.

Complex numbers receive surprisingly little mention in the *Using Matlab* manual. One surmises that they are a *struct*, but this is definitely not the case for sparse arrays (which have a special representation), and I have found no indication of how complex numbers fit into the type system.

The ClawZ tool works with a fairly radical simplification of this type system for the purposes of translating parameters, allowing in effect only real scalars, vectors and matrices, which map to Z types $\mathbb{R}$, $seq\ \mathbb{R}$ and $seq\ seq\ \mathbb{R}$. Probably the most complex which would be appropriate for ClawZ would be adding Complex and String into the expression types as follows:

{Scalar, Vector, Matrix} * {Real, Complex} + String.

8.1.2 Signal Types

Signal types are also discussed in section 3.

Once again there is no clearly documented type system, and so we must reverse engineer a type system as best we can.

Perhaps surprisingly it is not clear that the Simulink types are strictly a subset of the Matlab expression types.

A simple choice would be to base the signal type on the information available in the .rtw file, (also available through the graphic interface on pop-up windows), which is, the numeric type (allowing complex as a numeric type) and possibly also the signal width. The numeric types for our purposes can be simplified to {Real,Complex}, and Simulink signal types may then be represented as:

$$\{Real,\ \ Complex\} * \mathbb{N}$$

The Using Simulink manual does say that matrices are allowed as signal values, but I have been unable to find any trace of this, and I am guessing that this simply reflects the possibility that a Simulink block may treat its input vector as a matrix.

Two decisions to be made before choosing a representation for signal types are, firstly whether complex numbers are to be supported, and secondly whether the width of a signal is to be a part of its type, or just the scalar/vector distinction or not even that. In the documentation the terms scalar and vector are used, but they are used for "signal of width one" and"signal of width greater than one", there is no such thing as a unit vector. The Simulink type system can only be considered a subsystem of the matlab type system if scalar and vector signals are considered to have the same type, the term *scalar* being construed as *unit vector*.

From this point of view (and this is reflected in the Simulink user interface), the type of a signal is just its numeric type (complex, double, int8...), and for our purposes this would simplify to {Real, Complex}.

For the purposes of signal type inference ClawZ has now an internal representation of signal types. Under that representation a signal type may either be a scalar, a vector (possibly with some known width) or a bus structure (which is a labelled product of signal types). For fuller details see [1]. ClawZ signal type inference does not support complex signals at present.

8.1.3 Fcn Block Parameter Expressions

This is a special language allowing only scalar arithmetic. References to specific elements of the input vector or to arrays in the Matlab workspace are permitted, but not to whole arrays or slices.

All values are therefore either *Real* or *Complex* scalars. ClawZ does translate these expressions, only into expressions of type $\mathbb{R}$, but does no static analysis (none is needed).

8.1.4 The Z Types of Library Block Specifications

Early versions of the ClawZ translator placed no constraint on the types of the Z specifications of library blocks, and had no knowledge of these types. Consequently, ClawZ was unable to detect whether the translated diagram would be type-correct, and unable to use type-correctness as a constraint while selecting block specifications to instantiate.

If the purpose of static analysis of the diagram were to reduce the occurrence of type errors in the translation, then *either* there would have to be a known relationship between the type of a signal the type used to represent it in Z, *or else* the types of the available Z specifications would have to be known during static analysis and the analysis would have to be undertaken in terms of these types, to check compatibility of the blocks being wired together. In this latter case the signal types and widths obtainable from the .rtw file might act as a constraint on the allowable translations, but leave open whether a signal of length one is represented by a sequence type or not.

There is a third alternative, which is to allow the information in the .rtw file to be used in selection criteria specified in the metadata.

Information about the types of the ports on the Z specifications is now selectively made available in the library metadata. This information corresponds to the type system used for signal type inference in ClawZ, described above. Since this information is obtained as a result of a library match, not independently of the library, it cannot be utilised in library matching. It is used as a seed for a type inference phase the results of which are used during block virtualization and/or synthesis.

8.1.5 Matlab Operators and Functions in Z

8.1.6 Fcn Block Operators and Functions in Z

Assuming that all Fcn expressions have type *Real* or *Complex*, the type of the operators can be determined from their arity.

8.2 Sources of Information

There are a number of information sources, or possible future sources, which may be consulted to obtain the full content of a Simulink model. In this section these sources are identified and their contribution to the semantics is discussed.

The sources are:

- the Simulink .mdl file
- one or more Matlab .m files
- the Simulink block libraries
- ClawZ Z library metadata files
- ClawZ Z library Z specifications
- Supplementary metadata for Fcn and Matlab expression translation
- Supplementary Z library specifications for operators and functions
- a Simulink .rtw file

The primary information source is the Simulink model file, but this may contain Matlab expressions which mention matlab variables. The values associated with these variables may be defined in Matlab .m files, which must be taken into account when simulating or reasoning about the model.

We have no machine readable information source available to us covering the content of the Simulink block libraries. Some of this information is manually translated into the ClawZ library metadata files and the ClawZ Z Library specifications.

The Simulink .rtw file contains no essentially new information, being derived during the rtw build process from the model. However, various derivations are made in this process, and this may be the most convenient source for some kinds of information (e.g. signal widths or parameter types). The assignment of signal widths and types to signals would appear to depend upon a detailed knowledge of the characteristics of the library blocks.

8.3 Inferring Types

Inferring types of signals depends upon a detailed knowledge of the characteristics of the Simulink blocks. If this were to be done by ClawZ (as opposed to extracting the required information from the .rtw file), then this information would have to be supplied to ClawZ as additional metadata (or hard coded). There is no relevant information available to ClawZ in the existing metadata.

The required analysis of the Simulink libraries would be non-trivial and has not been undertaken.

Later versions of ClawZ do incorporate a type inference capability, using built in knowledge of a small number of blocks together with information supplied in ClawZ Z library metadata about the types of the ports on the supported library blocks.

8.4 Using Types

If signal types were available early enough in the translation they could be used in selecting Z specifications for translating the Simulink blocks. However, knowing the type of the signals does not suffice to establish the type of the relevant port in the Z specification, and consequently there is no guarantee that the Z specification resulting from the translation of a diagram would be type correct. This is because a signal whose width is 1 could be interpreted either as a scalar or as a unit sequence. If the interpretation differs at the two ends of the wire then a type error results.

To prevent this from happening it would be necessary to *always* map width one signals to the same type. That means, in effect, either adopting a unified value space or replicating all block specifications which take vector inputs with special cases providing scalars inputs (and all combinations of scalar/vector inputs would be required).

In recent versions of ClawZ incorporating type inference, the type inference is seeded by metadata which only becomes available for use after a match has been made between a Simulink library block and a corresponding Z library block. Type inference therefore takes place after library matching and information about port types derived from inference cannot be used in the library matching process. It would be possible to undertake an earlier seeding process using hard coded data about the Simulink blocks which might then enhance the matching process, but this is not a currently anticipated direction of development. Hard coded knowledge can be used in the type inference stage which follows library matching and the resulting information is then available for the virtualization and synthesis stages which now follow.

Virtualization and synthesis both use the available type information in ad hoc ways specific to particular block types. Usually virtualization and synthesis are inhibited for blocks whose input port types were not discovered during signal type inference.

8.5 Obtaining Z Specification Types

For the reasons indicated in the previous section, it is difficult to use signal type information to ensure a well type output specification. This can only be done without knowing the types of the Z specifications if a uniform convention is applied for the representation of signals of length 1.

However, if the types of the Z specifications are known, then this can factor in the selection algorithm. In this case however, it is not clear that knowing the signal types helps a great deal. You have to check the Z type at either end of a wire and select specifications which match, and having the inferred signal type doesn't really help with this.

To ensure type correct output there are three credible options:

1. Adopt a unified signal value space.
2. Manually include type information in the metadata.
3. Automatically extract type information from the ClawZ Z library.

Although item 2 is now implemented, it does not suffice to guarantee type correct output.

9 TRANSLATOR ENHANCEMENT OPTIONS

This section is now updated to reflect the status in the current version of ClawZ. Enhancements which are now partly or wholly implemented are shown as such.

An important consideration in the evaluation of prospective changes to the translator is whether the proposed change results in new output from an existing model.

For help in making decisions the following terms are used in the descriptions of the changes.

benign - changes which extend the range of models which can be translated without causing any change to the results of any currently translatable model

discreet - changes which might result in different output, but which are only invoked by some special action (e.g. extra information in the metadata) and which result in unchanged output where the new feature is not invoked.

disruptive - a change which might result in new output from an existing model (which currently translates automatically into type-correct Z), whether the user likes it or not.

It should be noted that benign changes may still impact QinetiQ if they are working by manual editing of the results of incomplete translations. However, if the recommended method of automating such changes is undertaken, the translation should become complete and will not be affected by a benign change.

The enhancements discussed fall into a number of groups as follows:

1. use of static analysis to aid diagram translation and block selection, reducing or eliminating type errors in the resulting specification
2. enhancements to the translation of parameters, permitting the translation of a wider range of Matlab expressions
3. virtual blocks and variable numbers of ports
4. single value space and complex numbers

9.1 Static Analysis

There are two distinct domains in which static analysis is relevant.

The first domain is that of Simulink signals. When viewing a model it is possible to arrange for signal type and signal width information to be displayed, either on the diagram or in pop-up windows. This information is also made available in the .rtw file. Clearly simulink is undertaking some limited static analysis to obtain this information.

However, in the broader domain of Matlab expressions, which may appear in Simulink models as parameters to blocks, and which are also used in creating the variable environment (possibly using .m files) in the context of which the parameters are evaluated, there appears to be little or no checking which is strictly static.

9.2 Signals

There are three more or less independent choices were considered this area.

- inference of simulink signal types v. inference of Z types
- inference by clawz v. use of .rtw file
- use for block selection v. use for signal coercion

A signal inference capability is now implemented. It works with a simplified type system devised for use in ClawZ, which reflects the types of ports in the Z specifications of blocks. The type information used as a starting point for this inference is taken from new parameters in the library metadata, knowledge of the relevant inference rules for a small subset of the Simulink block types is built into ClawZ. No reference is made to the .rtw file. The application for the resulting type information is neither of those anticipated, it is used by virtualization and synthesis facilities.

Further development in this area is most likely to consist of a piecemeal extension to the signal inference capability to encompass a larger subset of Simulink block types.

9.2.1 Block Selection Using Signal Type Inference

Signal inference is now undertaken after library matching seeded by information obtained from the metadata by the matching process. The information used in virtualization and synthesis. In principle the possibility remains to enhance block selection using type inference, but this would require moving forward the type inference stage and either finding another source of seed data or building the necessary information into ClawZ.

The trend of ClawZ developments has recently been to make more use of hard coded knowledge in ClawZ and less use of the library, and it therefore seems likely that the effects which might have been achieved by more sophisticated selection may instead be realised by wider coverage for virtualization and synthesis.

9.2.2 Block Selection Using .rtw Width and Type Information

There are really two options here.

Either ClawZ knows something about how signals are represented in Z and something about the types of the library blocks and constrains the block selection using this knowledge.

Or, you just make signal type and width selection criteria which can be specified in the metadata. That's much neater and more flexible.

Type information is now available in the metadata, and it is possible that selection could be enhanced by matching this against information in the .rtw file. The cost however would probably be high and the benefit marginal.

9.2.3 Automatic Signal Coercion

If signal types and Z specification types were known to the diagram translator, the translator could insert signal coercion functions under some circumstances.

If this were only done as a last resort, i.e. where a complete translation would not otherwise be possible, then it would be *benign*.

Coercions can be classified as either widening or narrowing, where conversion of scalar to vector or real to complex widens and the opposite narrows (there are mixed cases, e.g. complex scalar to real vector). Narrowing can be accomplished by using the inverse of the widening function, which will be undefined in some cases. This may be safe from the verification standpoint since reasoning about a narrowed signal will require demonstration that the value coerced is in the appropriate domain. However, the only case we would expect to arise would be the narrowing of a signal of width 1 from vector (sequence) to scalar, the legitimacy of which can be established by checking the width in the .rtw file.

I know no circumstance where narrowing from complex to real is appropriate. This would normally be accomplished by a block implementing the required conversion rather than by the diagram translation algorithm.

9.3 Expression Handling

The handling of Matlab expressions is one of those delicate areas in which the trade-off between obtaining accurate semantics and broad cover and holding down the complexity of the resulting Z is apparent.

Matlab is an interpreted imperative language. Fortunately, side-effects in Matlab expressions appear to play little or no role in the semantics of Simulink models, and the use of side effects in .m files setting up the variable environment are likely to be confined to variable assignments which can be interpreted as constant definitions.

Though ClawZ diagram translation makes no assumptions about the types used for signals, parameter translation makes assumptions about the representation of vectors and matrices as Z sequences. At present two parameter translation mechanisms are available which provide a choice of representation, but both presume that the number of dimensions in the matlab parameter values is no greater than 2 and that it is known whether the value is a scalar a vector or a matrix (even the "unified" parameter type requires this), even though this distinction is not clear dynamically in Matlab (let alone statically).

It does appear probable that the number of dimensions of a variable could be established statically. However, this is always at least two, and the distinction between a scalar, a vector and a matrix can only be based on the value of the dimensions for matrices. A scalar is a matrix with one row and one column, a vector is a matrix with only one row (excepting column vectors, which we need not count as vectors). This cannot always be established statically, but we can expect that it can be statically determinable for the subset of the Matlab language to be supported by ClawZ, or we can default to assuming non-trivial size for any dimension which is not known (this is likely to work by passing a unit vector instead of a scalar).

9.3.1 Dimension Tracking

In version 0.4.1 of ClawZ there was a loss of discrimination in selecting the library block when an explicit vector display is replaced by a variable defined in a .m file. A vector display would fail against a metadata block requiring a scalar input, but a variable name would not. Consequently, if a block type allowed either a scalar or a vector parameter type selection of the wrong Z specification might occur under these circumstances.

ClawZ does now undertake dimension analysis for matlab expressions translated in .m files, producing a data file which records the number and size of dimensions of each variable. This information is then used when translating matlab expressions which are the parameters of Simulink blocks. This gives a more accurate understanding of the type of parameters resulting in better library matching. This capability applies only to variables defined in .m files and could possibly be extended to mask variables.

9.3.2 Expression Parameter Translation

ClawZ does now translate a subset of matlab expressions either in .m files or when occurring as parameters to Simulink blocks. This is a small subset and could be extended if necessary.

9.3.3 Fcn Block Parameters

These are now translated automatically.

9.3.4 If Block Parameters

These are similar to but not identical to Fcn block parameters. Two new parameter types would permit these to be translated automatically (one for the *if* condition and one for the *elseif* conditions).

This is a *benign* enhancement.

9.4 Virtual Blocks and Variable Port Numbers

9.4.1 Opaque Virtual Blocks

Non-transparent virtual blocks can only be supported by special action in the diagram translator.

This has been done for Action ports, which now result in more complex translation of subsystems to capture the semantics of action subsystems. Other conditional subsystems are candidates for support in a similar manner but have not yet been implemented.

Typically extensions in this area appear to be *benign*.

9.4.2 Transparent Virtual Blocks

Transparent virtual blocks can be supported via the block library. Since variable numbers of ports are not supported this requires many different definitions in the Z library. Even if variable numbers of ports were to be supported this would still not cater straightforwardly for the full range of applications of Mux and Demux blocks.

ClawZ is now capable of fully virtualizing the main kinds of transparent virtual block, resulting in smaller and simpler subsystem schemas. This does not extend to subsystems, which are always treated as atomic (virtualization of a subsystem would be likely to widen the schema of the enclosing system or subsystem which may not be desirable).

ClawZ also virtualizes some transparent block types which are not considered by Simulink to be virtual. This means that instead of having a separate schema for the block, the function of the block is incorporated as an expression in the predicate of the schema for its enclosing subsystem.

9.4.3 Variable Numbers of Ports

Support for blocks with variable numbers of ports could be provided by:

1. Extending the metadata to provide information about whether variable numbers of ports are supported by a block and to determine the number of fixed ports.
2. Modifying the translator to refer to variable ports by indexing along a sequence.

Two additional parameters in the metadata would be appropriate: *FixedInPorts* and *FixedOutPorts* giving the maximum number of input and output ports respectively which would be individually mapped. All input and output ports above those numbers would be passed as a single input vector and a single output vector.

This enhancement would permit better ways of mapping blocks such as *Mux* and *Demux*, but these would not be invoked unless selected in the metadata. If *Mux* and *Demux* were handled directly by the diagram translator, there would still be plenty of other applications for variable numbers of ports (e.g. *Sum* and *Product*).

Though this enhancement is still viable and useful, the need for it is now mitigated by support for virtualization and synthesis in ClawZ.

9.5 Single Value Space and Complex Numbers

Though there is no present requirement for support of complex numbers, it is possible that they might be required in future.

At present ClawZ would allow some limited exploitation of complex numbers by extension of the library. It does not support complex valued expressions in parameters, but a variable given such a value in a .m file could be used as a parameter, and the .m file could then be manually translated. Though *ProofPower* does not support complex numbers, the introduction of a suitable type for complex numbers would be straightforward (though the theory and proof support would be more arduous).

Some of the enhancements under discussion, because they build more knowledge about types into ClawZ, would make matters worse if they were undertaken without consideration of complex numbers. For that reason it is arguable that even if complex support is not now required, such enhancements should make provision for it.

To enhance the existing ClawZ tool to support complex numbers would be primarily a question of extending the parameter and .m file translation to allow complex literals. The development of a theory of complex numbers would also be necessary if any reasoning were to be undertaken about specifications involving complex numbers.

Costs and impacts in this area are now greater as a result of the extra hard coding of translation introduced by virtualization and synthesis.