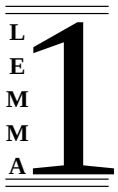


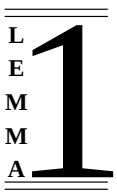
Lemma 1 Ltd.
c/o Interglossa
2nd Floor
31A Chain St.
Reading
Berks
RG1 2HX

ClawZ

User Guide

Version: 9.13
Date: 19 June 2003
Reference: DAZ/USR505
Pages: 51

Prepared by: R.D. Arthan
Tel: +44 118 958 4409
E-Mail: rda@lemma-one.com



0 DOCUMENT CONTROL

0.1 Contents

0.2 List of Figures

0.3 Document Cross References

[1] LEMMA1/DAZ/ZED503. *ClawZ - The Semantics of Simulink Diagrams.* R.B. Jones, Lemma 1 Ltd., `rbjones@rbjones.com`.

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

[3] LEMMA1/DAZ/ZED505. *ClawZ - Z Library Specification.* R.B. Jones, Lemma 1 Ltd., `rbjones@rbjones.com`.

0.4 Changes History

Previous released version: issue 9.11: accompanied ClawZ version 1.1.4.

Issue 9.13: accompanies ClawZ version 1.1.5; documents the new facilities that support references to blocks in artificial subsystems of a library.

0.5 Changes Forecast

As determined by review.

0.6 Trademarks

Simulink and MatLab are trademarks of The MathWorks Inc.

1 GENERAL

1.1 Scope

This document comprises part of the deliverables from the Real ClawZ project, placed by QinetiQ Malvern with Lemma 1 Ltd. It gives instructions for using the ClawZ tool developed by that project.

1.2 Introduction

ClawZ is a tool whose objective is to link the Simulink system with the ProofPower dialect of Z and, in particular, to provide a bridge between the use of Simulink to define control law diagrams and the use of the Compliance Tool component of ProofPower to specify and verify Ada code using Z.

ClawZ operates by translating a Simulink model into a Z specification. This Z specification may then be used in conjunction with a library of supporting definitions to construct a Compliance Argument which may then be formally verified using ProofPower. Figure 1 illustrates the main inputs and outputs of this process. In addition to translating Simulink models into Z, ClawZ also provides a facility to translate Matlab .m files containing definitions of variables into Z. Figure 1 illustrates the main inputs and outputs of the translation processes.

This document is the user manual for version 0.8.1 of ClawZ. The rest of this document is structured as follows:

Section 2	gives a more detailed technical overview of what ClawZ does based on a simple example model.
Section 3	describes the use of the ClawZ model translator to translate a Simulink model file into Z.
Section 4	describes the use of the ClawZ .m file translator to translate a Matlab .m files into Z.
Section 5	describes how the Z document produced by the model and .m file translators can be processed with ProofPower. It includes a complete compliance argument based on the example presented in section 2.
Appendix A	gives the proof scripts for the compliance argument given in section 5.
Appendix B	describes how the ClawZ library can be extended.
Appendix C	describes how the mapping from simulink to Z names can be controlled.
Appendix D	gives the SML type declaration for the run parameters to ClawZ.
Appendix E	describes a flag which controls the output encoding.

Some familiarity with Simulink, ProofPower and Standard ML is assumed in this document.

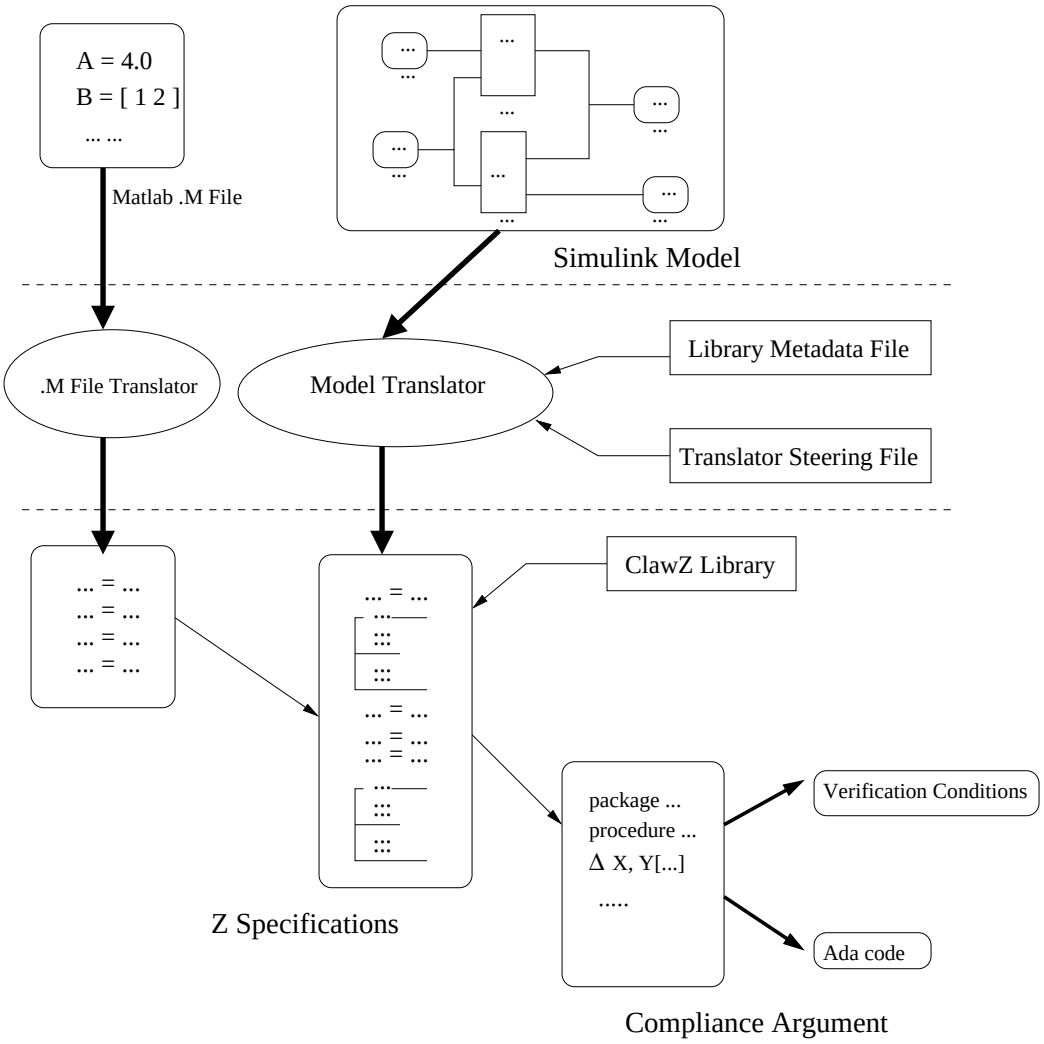


Figure 1: ClawZ Model Translation Processes

2 OVERVIEW OF ClawZ

2.1 A Simple Simulink Model

To give an overview of the operation of ClawZ, it is helpful to give a simple example of a Simulink model. Figure 2 shows a model comprising an input port, $In1$ and an output port $Out1$ connected by a unit delay block, *DelayBuffer*. This models a system which derives an output signal from an input signal. The input signal, I say, is sampled at regular time intervals[1] to give a sequence of discrete inputs $I_0, I_1, I_2, \ldots$. The output signal, O say, will then comprise the sequence $0, I_0, I_1, I_2, \ldots$. I.e., O is defined by $O_0 = 0$ and $O_i = I_{i-1}$ $(i > 0)$.

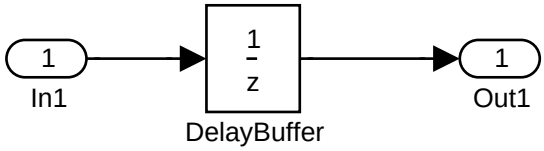


Figure 2: A Simple Simulink Model

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

2.2 The Model in Z

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

The first schema corresponds to the library block, *DelayBuffer*. The name, *unitdelay*, of the model has been used as a prefix for the schema name to reflect the position of the block in the hierarchic structure of the model. The schema $unitdelay\_DelayBuffer$ is defined in terms of a library function $UnitDelay\_g$ which we will discuss in more detail in section 2.3 below.

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

2.3 Overview of the ClawZ Library

As the simple example has shown, the semantics of Simulink library blocks like the unit delay block is carried into Z by a library of Z definitions. These definitions are typically generic functions, which

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

z

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

z

unitdelay

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

$Out1! = DelayBuffer.Out1!;$
$DelayBuffer.In1? = In1?$

Figure 3: The Simple Model in Z

when applied to appropriate arguments result in operation schemas following the usual Z conventions augmented with a special convention for handling initial values.

Figure 4 shows the definition of the library function $UnitDelay\_g$ that is used in the translation of our simple example. The function is generic with respect to the type, X, of the inputs and outputs of the unit delay block. In our example, $X = \mathbb{R}$. The function is parameterised by a binding giving the initial value of the state; applying the function to a particular binding, say $(X0 \mathrel{\widehat=} 0\ e\ 0)$, as was done in figure 3, results in a schema equivalent to the one shown in figure 5.

Informal Z

$[X]$

$\boldsymbol{UnitDelay\_g}: [X0: X] \to \mathbb{P}\ [In1?, initial\_state, state, state', Out1!: X]$

$\forall\ pars: [X0: X] \bullet$
$UnitDelay\_g\ pars = [In1?, initial\_state, state, state', Out1!: X \mid$
$\quad initial\_state = pars.X0$
$\quad \land\ Out1! = state$
$\quad \land\ state' = In1?]$

Figure 4: The Unit Delay Library Block in Z

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

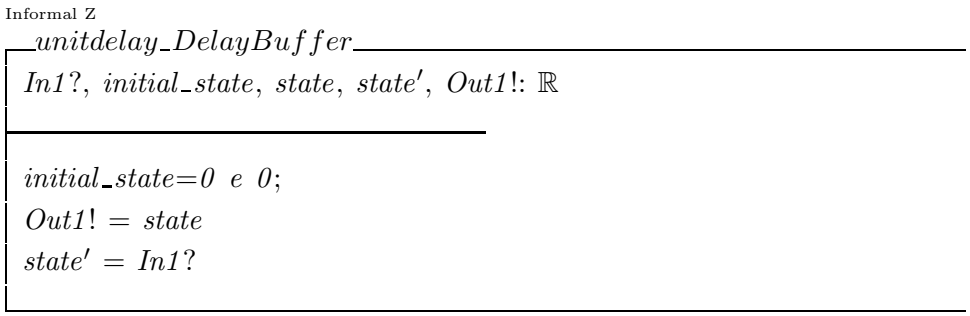


Figure 5: The Unit Delay Library Block Instantiated

3 USING THE MODEL TRANSLATOR

3.1 Files Used by the Model Translator

As suggested in figure 1, the ClawZ model translation process involves several different kinds of file. The input files are described in the following table:

File	Description
Model file	This is the file produced when you save a Simulink model or library and is your main input to the model translation process. The suffix "'.mdl" is normally used for this file, e.g., "unitdelay.mdl".
Library metadata file	This file contains a description of the library blocks supported by ClawZ in a format that allows the model translator to select the most appropriate Z translation for a library block. The library metadata file supplied with ClawZ is called "zed505.lmf". The content of this file is described in appendix B of this document and in the document ZED505[3]. If your model uses blocks that are not supported by the supplied library, you can extend the library yourself. See appendix B for more information on this.
Translator steering file	This file allows you to control the way the translator maps Simulink names to Z identifiers. It is optional: if you do not provide a translator steering file, then the translator will use a default algorithm for the mapping. More information on this file is given in section 3.4 below.

The output files are described in the following table:

File	Description
Z Output files	These are where the translator writes the Z translation of your model. You can arrange to write all the Z into a single file or to direct the Z for different parts of your model into different files. A recommended form for the name of these files is, for example, "unit-delay.zed.doc". The ".doc" extension makes it easier to run some of the ProofPower programs on the file while the ".zed" part reminds you that it isn't a complete ProofPower document, since it contains no LaTeX document-making commands. The output files are written by default in ProofPower ASCII format to make them easy to transport between systems. They may alternatively be written in the ProofPower extended character set, by setting a flag, see Appendix E. .
Library metadata output file	When translating the .mdl file for a Simulink library, ClawZ can produce metadata suitable for use in a subsequent run of ClawZ. The metadata permits block references referring to subsystems of the Simulink library from which it is derived.
Other output	Some important information is written to the standard output, which the user may redirect to a file of his choice. This includes warning and error messages, and information about the mapping from Simulink blockname paths to Z identifiers. Diagnostic information about inferred port types may be dumped to a nominated file, as described in section 3.3.

3.2 Running the ClawZ Model Translator

The ClawZ model translator is run by calling the Standard ML function `clawz_run`. The function requires as a parameter a Standard ML record of type *RUN_PARAMS* with four components to identify the various files used by ClawZ, and a fifth component to specify parts of the Simulink model that are to be treated as artificial subsytems. The ML declaration of this ML type may be consulted in Appendix D.

Before describing the record type we present details of some of its components.

3.2.1 Patterns

For various purposes it is necessary to refer to one or more of the Simulink blocks in a model. This is done using an ML value of type *string* as a pattern.

A pattern must conform to the following grammar:

```
pattern ::= <pat_el> ("/" <pat_el>)*
pat_el ::= "?" | "*" | <subsys-name>
```

Inside the $< subsys - name >$ parts of a pattern, you may use a backslash character as an escape character, if the subsystem name includes question marks, asterisks, slashes or backslashes. Question marks in a pattern act as wild-cards for a single subsystem name. Asterisks act as wild-cards for a sequence of subsystem names.

3.2.2 Output Filter Specifications

An output filter specification is an ML record of type *OUTPUT_FILTER_SPEC* which is used to specify which parts of a translated model are to be written to some specific file.

The components of the record are shown in the following table:

Name	Description
incl	a list of patterns for specifications to be written to the file
excl	a list of patterns for specifications NOT to be written to the file
file_name	an optional filename in the form *Value* $<$ *filename* $>$, in which case the selected output is written to that file, or *Nil*, in which case the output is written to the standard output.

"patterns" are described in section 3.2.1.

The specifications arising from translating a block in the Simulink model will be output to the designated destination if the hierachic name of the block is covered by one of the patterns in the "incl" list and not covered by any pattern in the "excl" list.

3.2.3 Block Modifier Specifications

When working with "artificial subsystems", "block modifier specifications" are used to specify the blocks that make up the artificial subsystem and to define which artificial subsystem of a library is to be used when translating a reference to a block in the library. This is done using an ML record of type *BLOCK_MODIFIER_SPEC*.

The fields in this record are as follows:

Name	Description
path	a pattern that uniquely identifies a block in the model
filter	If "path" identifies a subsystem, this gives a specification of which blocks are to be retained or excluded from the subsystem; if "path" identifies a library block, that library block must be a block reference and then "filter" gives the name of an artificial subsystem of the library to be used in the translation.

"patterns" are described in section 3.2.1.

The filter must be one of:

- the constructor "Include" followed by an ML list of strings which are the names of Simulink blocks;
- the constructor "Exclude" followed by an ML list of strings which are the names of Simulink blocks;
- the constructor "ASname" followed by a string giving the name of an artificial subystem of a library.

If the filter is constructed with "Include" or "Exclude", the block identified by the path must be a subsystem. If it is constructed with "ASname", then the block must be a block reference to a block in the indicated artificial subsystem of a library.

3.2.4 Artificial Subsystem Specifications

As well as providing a translation of a complete Simulink model, ClawZ can translate specified parts of a model, called "artifical subsystems", independently of the other parts of the model. Breaking down a model into artificial subsystems can help to make large models more manageable, e.g., by letting you view the model in a way which corresponds to the way in which a software implementation is organised. To describe an artificial subsystem an ML record of type *ART_SUBSYS_SPEC* is used.

The fields in this record type are as described in the following table:

Name	Description
`name`	The name to be given to the artificial subsystem, this will be used like a Simulink blockname for the top level of the artificial subsystem.
`top`	This is a block modifier specification (see section 3.2.3) for the top level subsystem of this artificial subsystem.
`rest`	A list of block modifier specifications. Each block modifier specification in this list identifies using a path relative to the top level subsystem either *(a)* a subsystem for filtering, together with a list of blocknames for inclusion, or a list for exclusion or *(b)* a library block that is a block reference to a block in a library, together with the name of an artificial subsystem of the library to be used when translating the block reference. Block modifier specifications are described in section 3.2.3.
`output_spec`	This specifies, in the format described in section 3.2.2, which files to use as the output files. It controls only the output of material specific to this artificial subsystem. Where the artificial subsystem contains blocks which are unchanged from the original model the original specification will be referred to. The pathnames used must correspond to paths in the artificial subsystem, i.e. the subsystem named as the top-level will have the name given to the artificial subsystem, and all contained subsystems will have paths constructed in the usual manner from that base-point.

All paths are specified in the same manner as described in section 3.2.1, i.e. as patterns, except that when described as "paths" they must uniquely identify a single Simulink block.

3.2.5 The ClawZ Run Parameters

Name	Description
`model_file`	The Simulink model file.
`output_spec`	This specifies which files to use as the output files. It must be a list of *OUTPUT_FILTER_SPEC*s as described in section 3.2.2.
`meta_file`	The name library metadata file. Use "zed505.lmf" unless you have extended the library yourself.
`steer_file`	The name of the translator steering file. The use of this file is optional: you may specify *Value* $<$ *filename* $>$, in which case $<$ *filename* $>$ is used for the translator steering file, or you may specify *Nil*, in which case the default name mapping algorithm is used unmodified (*Nil* is recommended for initial experimentation). See also appendix C.
`art_subsys`	A list of specifications of artificial subsystems of the model to be constructed by ClawZ and translated into Z. Artificial subsystem specifications are described in section 3.2.4.
`meta_output`	an optional filename in the form *Value* $<$ *filename* $>$, in which case the model is treated as a Simulink library and ClawZ outputs metadata for the blocks in the library to the named file, or *Nil*, in which case no metadata is generated. If a block appears inside an action system, ClawZ needs to generate additional Z paragraphs to give the semantics of the block when it is disabled and reenabled. If the metadata output filename is specified, ClawZ generates the additional Z paragraphs for all blocks processed, otherwise, the additional Z paragraphs are generated only for blocks which appear within action systems.

The most convenient way of working is to create a Standard ML source file containing one or more calls to this function. The following shows a simple source file to run ClawZ.

```
clawz_run {
        model_file = "unitdelay.mdl",
        output_spec = [{incl=["*"], excl=[], file_name=Value"unitdelay.zed.doc"}],
        meta_file = "zed505.lmf",
        steer_file = Nil};
```

Note that ClawZ will overwrite the output file (`unitdelay.zed.doc` in the above example) so make sure that the file does not contain any information you wish to keep before you run ClawZ.

If the above commands were stored in the file `test.ML`, you would run ClawZ from the UNIX command line as follows.

For an SMLNJ build of clawz:

```
sml @SMLload=clawz <test.ML
```

For a Poly/ML build of clawz:

```
poly -r clawz.polydb <test.ML
```

The *output_spec* component lets you distribute the Z translation of a model over several output files. As shown in the above example, the simplest output specification is to ask for all subsystems to be translated into a single output file using the wildcard "*" as the single member of the list of subsystems to be included.

If you have a system called "mysys" with top-level subsystems "subsys1", "subsys2" and "subsys3", you could separate the Z into three separate output files, one for each subsystem using the following output specification:

```
output_spec =
        [{incl=["mysys/subsys1/*"], excl=[], file_name=Value"myfile1.zed.doc"},
         {incl=["mysys/subsys2/*"], excl=[], file_name=Value"myfile2.zed.doc"},
         {incl=["mysys/subsys3/*"], excl=[], file_name=Value"myfile3.zed.doc"}]
```

As another example, if you wanted to create two output files, one to contain the system and its immediate subsystems, and another with all the subsystems further down in the hierarchy, then you could use the following output specification:

```
output_spec =
        [{incl=["*"], excl=["?/?/*"], file_name=Value"toplevel.zed.doc"},
         {incl=["?/?/*"], excl=[], file_name=Value"lowlevel.zed.doc"}]
```

3.3 ClawZ Diagnostic Output

ClawZ writes diagnostic output to the standard output channel. To capture this output in a file, you can use the output redirection facility of the UNIX shell.

For an SMLNJ build of clawz:

```
sml @SMLload=clawz <test.ML >test.log
```

For a Poly/ML build of clawz:

```
poly -r clawz.polydb <test.ML >test.log
```

In addition to the diagnostic output, the Standard ML compiler may write garbage collection diagnostics and other information to the standard error channel.

The model translation process has two main phases: a parsing phase when the model and other input files are read and checked and a semantic phase in which the model is analysed and translated.

At the end of the semantic phase, the Z translation is written to the output files and if requested metadata is generated and output.

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

The translator attempts to recover from certain error conditions during the semantic phase. One of the most common problems in this phase is when the translator encounters a use of a library block that it cannot translate. In this circumstance, it generates a warning message and continues, effectively ignoring the problematic block. Uses of the block will result in Z output that will not type-check because the Z translation of the block name will be used but not declared. This error recovery enables you to assess the completeness of coverage of the supplied library for your model and so decide whether it will be feasible for you either to extend the library to cater for your model or to select artificial subsystems in the model which do not give rise to problems and can usefully be analysed.

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

There are error conditions during the semantic phase which should not be considered as recoverable without careful inspection of the problem. An example is when an If block is used without its "Show Else" condition being turned on. These conditions are reported as errors rather than warnings and, by default, no Z output is produced for a model or artificial subsystem giving rise to such an error. This behaviour is controlled by a flag "inhibit_output_on_error" . To force the production of Z even when errors are detected, use the following command:

```
set_flag("inhibit_output_on_error", false);
```

Additional diagnostic information, which may help in understanding the operation of the block synthesis capabilities, may be requested by setting the string control "porttype_dumpfile" to the name of a file. ClawZ will the write information inferred about the signals on the ports in the model into that file. To set the value of this control you use the command `set_string_control`, prior to calling `clawz_run`. For example, to dump this information to the file `portinfo.txt`:

```
set_string_control("porttype_dumpfile", "portinfo.txt");
```

3.4 Translator Steering File

The user may supply a steering file which contains various information influencing the translation of the model.

The kinds of information which may be supplied in the steering file are as follows:

Name Mapping Information This allows the user to override the translation of specific Simulink blocknames into Z.

Port Type Information This allows the user to supply information about the types of signals in the model which ClawZ is not able to deduce itself. This information is needed for ClawZ to undertake block synthesis.

Matlab Variable Types This allows the types of matlab variables defined in .m files to be supplied to ClawZ. The information is generated by the .m file translator for inclusion in the steering file.

The syntactic structure of the steeing file is similar at the top level to that of a Simulink model, i.e. it consists of a sequence of named structures where the structure is contained in curly brackets. In all cases the structure consists of a sequence of name/value pairs (also as in a Simulink model, except that in some cases the "name" may be a complete Simulink path.

It is always permitted to split information of a single kind across multiple structures with the same name in a steering file, so that a steering file formed by concatenating several files of relevant information will be accepted. This is relevant, for example, when type information obtained by translating several .m files is required for the translation of a model.

3.4.1 Name Mapping Information

The translator steering file allows you to override the model translator's algorithm for mapping Simulink names into Z (see also appendix C). The format of this file is as shown in the following example

```
NameMapping {
        UnitDelay             "int517a/Unit Delay"
        DiscreteTransferFcn   "int517a/Discrete\nTransfer Fcn"
        ZeroOrderHold         "int517a/Zero-Order\nHold"
}
```

Each line in the body of the file contains a Z name followed by the full Simulink path of a block in the Simulink model, with subsystem names separated by "/" characters, enclosed in quotation marks. The idea is that if you specify an association between a Z name and a Simulink name in this file, the translator will use the association you have specified instead of its default algorithm for that Simulink name. The name used will be used as the local name for the block in question in the subsystem schema for the substem in which that block occurs. Global names for schemas are compounded systematically from the local names of the blocks in the full Simulink path of the block using separators which may be specified by the user. ClawZ imposes some constraints on the form of local Z names in order to ensure that the method of compounding local names translating the Simulink names in a path yields a unique Z name for each path. These contraints are applied to names supplied by the user in a `NameMapping` table and names not conforming to the constraints will be rejected. Fuller details may be found in appendix C.

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

3.4.2 Port Type Information

ClawZ can generate the Z specifications for some Simulink block types automatically. This is referred to as "block synthesis".

In order to undertake synthesis ClawZ may need information about the structure of the signals carried on lines connected to the block to be synthesized, but it may not be able to determine the required information by analysis of the Simulink model. One case in which ClawZ is unable to make a determination is on lines which are inputs to the system as a whole.

To enable synthesis to function where ClawZ would otherwise be incapable of inferring sufficient information about signal structures, a facility for the user to supply this information is provided. The information supplied is similar to that provided in the library metadata using port type parameters (see appendix B section B.3), but instead of being associated with a kind of library block, it is associated with a specific block in a Simulink Model.

Two structures in the steering file are supported for this purpose, `InPortTypes` and `OutPortTypes`, which are used for supplying port type information for input and output ports respectively. They have the same internal syntactic structure, which consists of a list of path/type-info pairs. The path should be presented in the same format as for the metadata `BlockPath` parameter, except that it must not be quoted and may be terminated either by unescaped white space or by an unescaped double quote character. If the path does not begin with an alphabetic character then the first character should be preceded by backslash. Whitespace characters or the double quote symbol occurring in the path must be escaped with a backslash. Wildcards are not supported in these paths. The type information should be in the same format as for the metadata `InputPortTypes` parameter (see section B.3).

An example of the required format is as follows:

```
InPortTypes {
        int517a/Mux                 "V2V3SSV"
        int517a/BusConstructor      "B6["
                                            "firstsignal:V2,"
                                            "secondsignal:S,"
                                            "thirdsignal:B3["
                                                    "one:S,two:S,three:S"
                                    "]]"
}
```

3.4.3 Matlab Variable Types

Matlab variable type information may be included in the steering file in one or more structures named *VariableTypes*. The required information is output by the .m file translator and need not therefore be understood by the user.

It consists of a sequence of name/value pairs. The name is the Z translation of the name of the matlab variable (differing from the original in possibly having prefix or suffix added as described in Appendix C). The value is a vector giving the dimensions of the variable, consisting of a comma separated list of non-negative integers enclosed in square brackets. For a scalar variable the empty list should be supplied ("[]"), and it is permitted to supply a zero for a dimension if its size is unknown (the .m file translator is not always capable of determining the dimensions, in which case the user might edit the .m file translator output to supply the information). Neither the name nor the type information should be enclosed in quotations.

3.5 Generating Metadata

When Simulink users create libraries and use these libraries in their models, they can translate the libraries into Z using ClawZ and generate metadata suitable for accessing these libraries at the same time.

Translating a library into Z is done in exactly the same way as translating a model, by passing the library .mdl file to ClawZ. Metadata will automatically be generated if the user gives a filename for the metadata using the *meta_output* field of the ClawZ *RUN_PARAMS* (see section 3.2.5). When translating a model which makes use of the library by block reference, the generated metadata should be supplied to ClawZ (in a single file together with any other metadata needed for the translation). Metadata is generated only for *subsystems* in the library (not for any other kind of block), and the metadata file will therefore suffice only for block references to subsystems in the library. If artificial subsystems of the library have been specified, then the generated metadata will include block specifications allowing subsystems of the artificial subsystems to be referenced from other models (see section 3.2.3).

Special considerations apply where a Simulink library contains an internal block reference, i.e. a block reference to a block in the same library. In this case metadata will be needed to resolve the block reference, but will not be available until after the translation. The translation should therefore be done twice, discarding the Z from the first attempt but using the generated metadata on the next translation. In exceptional circumstances this process may have to be repeated, since the metadata generated for a subsystem when a block reference inside that subsystem cannot be resolved may itself be incomplete. If the metadata is incomplete in this way then this will result in a free variable in the resulting Z and will be detected when the specification is type-checked. The translation should be iterated until no spurious free variables occur.

Where it is required to use by block reference blocks in a library other than subsystems, it may be possible to manually write appropriate metadata to acheive the desired effect.

It should be noted that when using block references to translated libraries the blocks referred to

should not be altered by any other means than changes to the actual values of the mask variables of the blocks referred to.

3.6 Block Synthesis and Virtualization

ClawZ can "synthesize" or "virtualize" some Simulink block types. Synthesis of a block means that the ClawZ model translator generates the Z specification for the block automatically rather than using a specification from the ClawZ Z library. Virtualization of a block means that the semantics of the block are translated directly into Z predicates at the points of use. In both cases this means that the semantics of the block is built into the ClawZ model translator rather than being described in the ClawZ Z library.

If an instance of a block does not match a specification in the ClawZ Z library, and the "virtualize" flag is set, ClawZ will attempt to secure the effect of the required block by including an appropriate expression in the predicate of the subsystem in which the block occurs instead of including a reference to a library block. If there is no library match and if the block has not been virtualized (either because the *virtualize* flag is not set, or because for some reason virtualization was not possible) then ClawZ will attempt to synthesize a Z specification for the block. A synthesized specification will be similar in character to a library specification and will be referred to in a similar manner (so the schema for the relevant subsystem will be similar to what it would have been had a library reference been undertaken). Synthesis is in principle more broadly applicable than virtualization (which is only applicable to blocks which are pure mathematical functions without any "state"). Virtualization has the advantage that it decreases the number of components in the signature of the subsystem schemas.

Virtualization and synthesis is applicable to certain block types as shown in the following table:

Block Type	**Virtualize**	**Synthesize**
Bus Constructor	Yes	Yes
Bus Selector	Yes	Yes
Constant	Yes	Yes
Demux	Yes	Yes
Mux	Yes	Yes
Selector	Yes	Yes
Terminator	Yes	Yes
Merge	No	Yes

A match against the ClawZ Z library takes precedence over virtualization, if a match is found virtualization will be inhibited. If virtualization is preferred the metadata permitting the match must be removed. If virtualization is not desired then suitable metadata for a match must be supplied, or else the "virtualize" flag should be set to "false", in which case synthesis will be attempted.

Either a match against the ClawZ Z library or successful virtualization will take precedence over synthesis, if either a library match or virtualization is sucessful synthesis will not be attempted. For sucessful synthesis there must be no match in the metadata and the virtualize flag must be set to "false" (unless the relevant block falls outside the virtualization capability).

To change the value of the virtualize flag the command `set_flag` should be executed prior to calling `clawz_run`, e.g.:

```
set_flag("virtualize", true);
```

The flag defaults to 'false".

3.7 Action Subsystems

An action subsystem is a system whose execution is conditional on an *If* or *Switch Case* block. The outputs of action subsystems are normally connected to merge blocks which pass on to their output from whichever of their inputs comes from an enabled subsystem. ClawZ is able to translate systems containing action subsystems subject to the following restrictions:

1. Each action subsystem must have only one output port.
2. The output of an action subsystem must go only to a *Merge* block.
3. All inputs to *Merge* blocks must come from action subsystems.
4. Action subsystems controlled by the same *If* or *Switch Case* block must feed the same *Merge* block.
5. Action subsystems feeding the same *Merge* block must be controlled by the same *If* or *Switch Case* block.
6. If an *If* block does not have its else condition turned on, the InitialOutputs parameter of the corresponding *Merge* block must not be specified as an empty vector.
7. If a *SwitchCase* block does not have its show default condition turned on, the InitialOutputs parameter of the corresponding *Merge* block must not be specified as an empty vector.

When checking restrictions 2, 3, 4 and 5 above, ClawZ does not look across subsystem boundaries. I.e., the checks will fail if the relevant blocks and subsystems do not all appear at the same level in the Simulink diagram. If any of the checks fail, Z output for the model or artificial subsystem will normally be inhibited (see section 3.3 for information on how to control the behaviour when errors are detected).

The Z translation of an action subsystem (or a subsystem contained in an action subsystem) involves several schemas whose names are derived from the name of the subsystem using suffixes, which by default are subscript lower-case letters. Section C.6 below describes how you can change these suffixes.

4 USING THE .m FILE TRANSLATOR

4.1 Files Used by the .m File Translator

The ClawZ .m file translation process involves three types of file. These are described in the following table:

File	Description
.m file	This is Matlab .m file (created with the Matlab editor or any text editor) containing the Matlab statements to be translated. The suffix "'.m" is normally used for this file, e.g., "mfile_eg.m". The purpose of the .m file translator is to translate Matlab variable definitions of the form "name = expression" into Z abbreviation definitions. Other types of Matlab statement in the .m file are ignored. The expressions on the right-hand side of the Matlab definitions must be either simple variables or numeric literal constants or be vector or matrices whose constituent scalars are simple numeric literal constants.
Output file	This is the file where the .m file translator writes the Z translation A recommended form for the name of this file is, for example, "mfile_ eg.zed.doc". The ".doc" extension makes it easier to run some of the ProofPower programs on the file while the ".zed" part reminds you that it isn't a complete ProofPower document, since it contains no LaTeX document making commands. The output file is written in ProofPower ASCII format to make it easy to transport between systems. To convert it to the ProofPower extended character set, use the program "`conv_extended`" supplied with the ProofPower document preparation package.
Variable types file	Into this file the .m file translator writes information about the types of the variables defined in the .m file. This is for use when translating models which make use of the variables, and is then supplied to the ClawZ model translator in the steering file.

4.2 Running The ClawZ .m File Translator

The ClawZ .m file translator is run by calling the Standard ML function `m_file_run`. The function takes as its parameter a Standard ML record of type *M_FILE_RUN_PARAMS* (see appendix D). These records have three components to identify the files and other information used by the .m File Translator, as shown in the following table:

Name	Description
`m_file`	The name of the .m file to be translated.
`output_file`	The output file. This is optional: you either specify *Value* $<$ *filename* $>$, in which case $<$ *filename* $>$ is used for the output file, or your specify *Nil*, in which case the standard output channel is used. The output file will be overwritten by the Z specification so make sure the file you specify does not contain any important information.
`parameter_type`	This parameter is a Standard ML string which selects one of several methods for translating the expressions in the .m file into Z. The value to use depends on the way the Z specifications for the ClawZ library represent the Matlab type system. The value of this parameter should be `"SVM"` for the library as currently provided.
`types_file`	The variable types file. This is optional: you either specify *Value* $<$ *filename* $>$, in which case $<$ *filename* $>$ is used for the output file, or your specify *Nil*, in which case no type information is output. The file will be overwritten so make sure the file you specify does not contain any important information.

The most convenient way of working is to create a Standard ML source file containing one or more calls to this function. The following shows a simple source file to run the .m file translator.

```
mfile_run {
       parameter_type="SVM",
       m_file="mfile_eg.m",
       output_file=Value "mfile_eg.zed.doc",
       types_file=Value "mfile_eg_types.txt"};
```

Note that the .m file translator will overwrite the output files (`mfile_eg.zed.doc` and `mfile_eg_types.txt` in the above example) so make sure that the file does not contain any information you wish to keep before you run the .m file translator.

If the above commands were stored in the file `test_mf.ML`, you would run the .m file translator from the UNIX command line as follows:

For an SMLNJ build of clawz:

```
sml @SMLload=clawz <test_mf.ML
```

For a Poly/ML build of clawz:

```
poly -r clawz.polydb <test_mf.ML
```

4.3 .m File Translator Diagnostic Output

The .m file translator writes diagnostic output to the standard output channel. To capture this output in a file, you can use the output redirection facility of the UNIX shell as described in section

3.3.

In addition to the diagnostic output, the Standard ML compiler may write garbage collection diagnostics and other information to the standard error channel.

If it encounters constructs in the .m file that it cannot translate, the .m file translator will output a warning message to the standard output channel and skip past the unrecognised construct. The diagnostics on the standard output channel includes a record of what parts of the input have had to be skipped.

5 PROCESSING THE TRANSLATED MODEL

5.1 Document Preparation

Once you have translated your model you can use the ProofPower tools to process the output file. You can run `doctex` to generate a LaTeX file which may be included in another document with the `\include` LaTeX command. For example, the contents of figure 3 were included in this document by first executing the UNIX command:

```
doctex unitdelay.zed
```

The LaTeX command used to generate the Z figure was:

```
\include{unitdelay.zed}
```

The `conv_extended` program lets you convert the Z document into the ProofPower extended character set. This enables you to view and edit the document on the screen in a more readable format using the `xpp` editor.

5.2 Type Checking

The Z document can be loaded into ProofPower in the usual way (using `docsml` from a UNIX shell to generate a `.sml` file and then using `use_file` in ProofPower to process the result). However, the `.doc` file can in fact just be loaded directly with `use_file` (since it does not contain any LaTeX commands or other material that cannot be processed by ProofPower directly).

The Z document generated by ClawZ depends on the ClawZ library which contains the schema definitions that give the semantics for Simulink library blocks and on the ClawZ toolkit which supports the schemas in the ClawZ library. To satisfy these dependencies you will need to load three files into a ProofPower-Compliance Tool database. The files needed are in the `doc` subdirectory of the release directory and are called `dtd528.sml`, `imp528.sml` and `zed506.sml`. Assuming you are working in the ClawZ installation directory, the following UNIX commands will create a ProofPower database called `clawzlib` in which you can type-check and reason about a Z document generated by ClawZ:

```
pp_make_database -f -p daz doc/clawzlib
pp -d clawzlib -f doc/dtd528.sml
pp -d clawzlib -f doc/imp528.sml
pp -d clawzlib -f doc/zed506.sml
```

The commands to create a Compliance Tool script to hold the translated model and the Compliance Argument to be based on it might be as follows:

SML

```
new_script{name = "unitdelay", state = initial_cn_state};
new_parent"CLT";
use_file"unitdelay.zed.doc";
```

5.3 Interfacing with a Compliance Argument

In this section we give a Compliance Argument for an implementation of our simple example. Our plan is to implement the unit delay model using the an Ada package along the following lines (in which, for simplicity, we have made the state variable global).

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

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

Z

$$UnitDelayProgVars$$

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

Now we define the interface schema:

Z

$$UnitDelayInterface$$

$$unitdelay;$$
$$UnitDelayProgVars$$

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

We can now give the formal specification of the package, in which we existentially quantify over the schema *unitdelay* so that only the program variables appear free in the post-condition.

Compliance Notation

```
package UnitDelay is
    delay_buffer : float;
    procedure step(sig_in : in float; sig_out : out float)
    Δ DELAY_BUFFER, SIG_OUT [∃ unitdelay• UnitDelayInterface];
end UnitDelay;
```

Finally, we give the formal specification of the package body. The Compliance Notation requires us to begin a new script for this.

SML

```
        new_script{name = "unitdelay_body", state = get_cn_state()};
```

The package body can now be given:

Compliance Notation

```
package body UnitDelay is
    procedure step(sig_in : in float; sig_out : out float)
    Δ DELAY_BUFFER, SIG_OUT
        [DELAY_BUFFER = SIG_IN ∧ SIG_OUT = DELAY_BUFFER_0]
    is
    begin
        sig_out := delay_buffer;
        delay_buffer := sig_in;
    end step;
begin
    Δ DELAY_BUFFER [ ∃ unitdelay• DelayBuffer.initial_state = DELAY_BUFFER ]
end UnitDelay;
```

To complete the package we have only to provide the package initialisation statements that refine the last specification statement:

Compliance Notation

```
⊑ delay_buffer := 0.0;
```

The compliance argument is now complete. It generates 4 VCs. For completeness, the script to prove these VCs is given in appendix A below.

A PROOFS OF THE VCS

We are still working in the theory containing the four VCs from the compliance argument of section 5.3. We now prove these VCs. First we use the Compliance Notation proof support tools to set up a proof context in which to work.

SML

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

Now we begin the proofs.

SML

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

ProofPower Output

```
Now 1 goal on the main goal stack

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

(* ?⊢ *)  ⌜true ⇒ true⌝
```

This trivial VC arises from the empty pre-conditions and is solved just by stripping.

SML

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

SML

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

ProofPower Output

```
Now 1 goal on the main goal stack

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

(* ?⊢ *)  ⌜∀ DELAY_BUFFER, DELAY_BUFFER_0 : FLOAT;
            SIG_IN : FLOAT;
            SIG_OUT : FLOAT
          | true ∧ DELAY_BUFFER = SIG_IN ∧ SIG_OUT = DELAY_BUFFER_0
          • ∃ unitdelay • UnitDelayInterface⌝
```

This VC is ensuring that the program-oriented post-condition used for the procedure *step* in the package body meets the post-condition derived from the Simulink model as used in the package specification. We apply the usual simplifications for Compliance Notation VCs (which include rewriting

with all applicable definitions in our present proof context) and then strip the goal. We then have an existential goal with an obvious witness: the only possible binding that meets the constraints of the interface schema. After supplying this witness, we have only to expand with the definition of $UnitDelay\_g$, which requires a little work because it is generic.

SML

```
a(cn_vc_simp_tac[] THEN REPEAT strip_tac);
a(z_∃_tac⌜(In1? ≙ SIG_IN,
        DelayBuffer ≙ (
                state ≙ DELAY_BUFFER_0,
                state′ ≙ DELAY_BUFFER,
                In1? ≙ SIG_IN,
                Out1! ≙ SIG_OUT,
                initial_state ≙ 0 e 0),
        Out1! ≙ SIG_OUT)⌝);
a(cn_vc_simp_tac[] THEN asm_rewrite_tac[z_gen_pred_elim[⌜U ⊕ ℙℝ⌝]cn_UnitDelay_g_thm]);
val vcUNITDELAYbody_2 = save_pop_thm"vcUNITDELAYbody_2";
```

SML

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

ProofPower Output

```
Now 1 goal on the main goal stack

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

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

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

SML

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

SML

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

ProofPower Output

```
Now 1 goal on the main goal stack

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

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

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

SML

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

That completes the proofs for the unit delay example.

B EXTENDING THE LIBRARY

To extend the ClawZ library, you provide additional block specifications to be fed into the model translator as library metadata and write Z paragraphs to support the resulting translated models. In this appendix, we give a simple example of this task and then provide more detailed information about the design of the library and how and when to extend it.

If your model contains block references to blocks in a Simulink library then you must provide metadata describing those blocks. This can be obtained from ClawZ by specifying a file for metadata output when running ClawZ to translate the library, or it may be compiled manually if for some reason automatic production is not feasible.

Block references into a library which has been compiled by ClawZ are only supported when they refer to subsystems in the library, and transmitted parameters are allowed only if the subsystem is a masked subsystem and must then correspond to the mask variables. If the ClawZ capabilities for translating and referring to libraries are insufficient for a particular model then a successful translation might still be possible by selectively or completely breaking the library links before translation.

If multiple sources of metadata are to be used they should be concatenated into a single file for ClawZ. If such a file contains multiple entries which match a single instance of a Simulink block then the latest in the file will be used, priority is therefore given to the metadata in the last of a concatenated set of metadata files. So if your metadata is intended to supply a customised treatment of a kind of block that is already supported in the supplied library metadata file, you can simply append your custom metadata to the suppled metadata file.

B.1 A Simple Example

Figure 6 shows a Simulink model that includes a Digital Clock block. If you process this model using the library metadata file `zed505.lmf` that is supplied with ClawZ, ClawZ will report the following error message because the library does not support this type of block:

```
Error near line 124: proc_lib_blocks/proc_lib_block: this use of this library
block is not supported (name: "Digital Clock"; type: "DigitalClock")
```

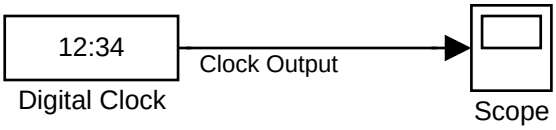


Figure 6: Digital Clock Example

Extending the library to add support for the Digital Clock block involves two main steps: we must design a Z specification to provide the semantics of the Digital Clock and prepare a library metadata entry to define the interface of that schema to the model translator.

The Digital Clock block has a parameter giving the sample time for the clock. If we make the design decision to have the translator pass this parameter through to the Z specification, we will need a Z paragraph along the following lines to implement the block:

z

$$
\begin{array}{|l}
DigitalClock : [SampleTime{:}\mathbb{R}] \rightarrow \mathbb{P}[initial\_state,\ state,\ state',\ Out1! : \mathbb{R}] \\
\hline
\forall\ pars{:}\ [SampleTime{:}\mathbb{R}] \bullet \\
DigitalClock\ pars = [\\
\quad initial\_state,\ state,\ state',\ Out1! : \mathbb{R} \\
\mid \quad initial\_state = 0\ e\ 0 \\
\land \quad state' = state +_R real\ 1 \\
\land \quad (\exists t{:}\mathbb{N} \mid real\ t *_R pars.SampleTime \leq_R state' <_R real\ (t{+}1) *_R pars.SampleTime \\
\quad \bullet\ Out1! = real\ t *_R pars.SampleTime) \\
]
\end{array}
$$

Following the format described in section B.3, we describe this in the Library metadata file as follows:

Text dumped to file usr505.lmf

```
BlockSpecification {
  Zname                    DigitalClock
  SelectionParameters {
    BlockType                    DigitalClock
  }
  TransmittedParameters {
    SampleTime                   Scalar
  }
}
```

To use the library with the Digital Clock extension, you add the above text to a copy of `zed505.lmf` in the file `mylib.lmf` say and then run the model translator giving `mylib.lmf` as the `meta_file` component of the parameter. The resulting Z translation is shown in figure 7.

B.2 Representing Simulink Blocks in Z

B.2.1 Use of Schemas

The translator requires all specifications to be prepared as Z schemas, or, in the case of parameterised specifications, as functions which yield Z schemas.

Where other considerations (such as the need for overloading) do not prevent it, the schema name used in the Z specification should be the same as the Simulink *blocktype*.

z

$$digclock\_DigitalClock \mathrel{\widehat=} DigitalClock\ (SampleTime \mathrel{\widehat=} 1\ e\ 0)$$

z

$$digclock\_Scope \mathrel{\widehat=} Scope$$

z

digclock

$$DigitalClock : digclock\_DigitalClock;\ Scope : digclock\_Scope$$

$$Scope.In1? = DigitalClock.Out1!$$

Figure 7: Digital Clock Example in Z

B.2.2 Port Naming Conventions

The names for the input output components of the schema, which are to correspond with the ports on the library block, must follow set conventions. These conventions are necessary to enable the names to be deduced from the information provided in the Simulink models. The information provided in the models is simply the port type (i.e. input or output) and the port number.

Input ports should be given the name "Inx?", where x is the number of the input port in question. Output ports should be named "Out$x$!", similarly. Trigger and Enable ports should be named "Trigger?" and "Enable?" respectively.

In the supplied library a special convention has been adopted for including components which specify the initial values of components of the state of the system. This is to use the prefix *initial_*, as described in section B.2.6. Initial state components have been included in all cases, irrespective of whether anything is known about the initial state.

B.2.3 Overloading

Many of the Library blocks provided by Simulink cannot be specified using a single schema because of the constraints imposed by the Z type system. For example, in very many cases the number of ports on an instance of a library block varies according to selections made at the time of instantiation.

It therefore necessary to treat this as a kind of *overloading*. Since Z does not support overloading, the translator supports the selection of one of a number of Z specifications using the parameters of the instance of the library block used in the model to decide the appropriate definition. This

provision for overloading can also be used to restrict the application of a single definition, where that specification covers only some of the possible uses of the relevant Simulink library block.

When undertaking the specification of a Simulink library block in Z it is necessary first to decide whether the full range of uses of the block are to be encompassed in the specification, and then to divide the range of intended applications into a number of cases each of which can be accomodated by a single Z specification (possibly parameterised, see section B.2.4).

Each of these cases should be specified separately and given its own name, which we recommend should begin with the name of the relevant Simulink blocktype and be differentiated by some suitable suffix (if more than one is required).

The selection of the Z specification to be used for any particular instance of a Simulink library block in a model is made on the basis of information supplied in a library metadata file which is prepared at the same time as the Z specifications for the Simulink blocks. This metadata is matched against relevant information in the model file.

B.2.4 Parameterisation

Parameterisation permits information entered by the Simulink model designer into the dialogue box when instantiating a library block to be passed to the Z library specification invoked in the translation of the model.

Parameters are passed by binding them together using an explicit Z binding display and passing this value to one of the functions specified for this particular library block. The person writing the Z specification of the library block must inspect the dialogue box and decide which of the values there should be passed as parameters. Values on the dialogue box not passed as parameters can be used in selection criteria to resolve overloading, e.g. the integration method in the discrete integrator could be used to select an appropriate specification rather than being passed as a parameter to a single specification.

If a specification is being written for a subsystem block in a library, which will be included in models by block reference, then it must accept a parameter which has one component for each of the variables masked on the target subsystem and one for each variable masked in some enclosing subsystem (in the library) and used in the target subsystem (or one of its subsystems). However, in most cases specifications for such library subsystems can be generated using ClawZ, in which case the correct parameterisation will be done automatically.

B.2.5 Discrete Operations

The Z specifications which form the library typically follow the usual conventions for the representation of operations by Z schemas. In particular the following conventions for decorating the names of schema components to indicate their role are used:

- Inputs are decorated with "?".

- Outputs are decorated with "!".
- The name of the component representing the next value of a component of the state of the operation is formed from the name of the state component by decoration with " ′".

B.2.6 Initial State

There is no standard convention for specifying the initial state of operations in a Z specification. Often a schema is defined giving the initial state, the name of which is related to that of the schema defining the state, e.g. a schema named *INITIAL_STATE*. This approach is not convenient for use in the library, where no separate schema is defined for the state of the operations, and where it is desirable to compose the complete definition of blocks into larger blocks.

For specifying initial state the supplied library uses a special convention in which the initial value of each individual component of the state of an operation is included as a separate component of the schema defining the operation. Thus, in the simplest case where there is a single component of a discrete operation representing the state, called *state*, there are three variants of this in the schema:

state the current value of the state

$state'$ the next value of the state

initial_state the initial value of the state

This information about the initial state is propagated through diagram definitions generated by the diagram translator. It is straightforward to separate this out from the final definition of the system, or to hide these intial state components.

B.2.7 Library Block State inside Action Subsystems

If a library block has internal state and is to be used inside an action system, the Z schema that defines its normal, enabled, behaviour, needs to be supplemented by two extra Z specifications to define the behaviour when it is disabled, and thereby to determine the value of the state when the block is next enabled. In the Simulink model the value of the state when execution resumes is defined by the parameter of the relevant action port, which can be set to "held" (meaning that the state is preserved during the disabled period) or "reset" (meaning that the state is reset to its initial value). A supplementary Z specification is required for each of these two cases.

The "held" and "reset" schemas are expected by the translator to have the same signature and the same parameters as the schema or schema-yielding function that defines the behaviour when enabled. The names of these two schemas are given in the metadata for the library block. By convention, in the supplied library subscripts "$_h$" "$_r$" are used to form the names, which follows the default naming convention used by ClawZ when generating similar Z specifications from model or library files.

B.3 Preparing Extensions to the Library Metadata

From the analysis of the Simulink library blocks to be supported the library specifier should have arrived at the following information which determines the mapping of instances of Simulink blocks onto Z specifications.

The information should consist of a number of *cases*.

With each case some or all of the following information may be supplied:

artificial subsystem name the name of an artificial subsystem of a library whose Z translation provides the Z specification to be used.

target specification name the name of the Z specification which should be used for instances of Simulink library blocks falling under the present case (either when they are not in an action system or to define the behaviour when inside an action system which is enabled).

held specification name the name of the Z specification which should be used for instances of Simulink library blocks falling under the present case to define the behaviour while disabled inside an action system whose action port parameter has the value "held".

reset specification name the name of the Z specification which should be used for instances of Simulink library blocks falling under the present case to define the behaviour while disabled inside an action system whose action port parameter has the value "reset".

block path pattern this is a pattern which will be matched against the full hierarchic name of the block instance for which a target specification is sought, a block falls under the present case only if the match is successful

port type parameters giving information about the types of signals expected on the ports of the block

used local variables giving a list of local variable used in the block but not explicitly passed to it

selection parameters a number of simulink block parameter names and values whose presence is an indicator that the case in question is applicable (this would normally *include* the Simulink blocktype)

transmitted parameters a number of parameter names and translation codes which indicate parameters to be passed to the Z specification

The information for each case must include the target specification name and at least one selection parameter.

An instance of a Simulink Library block falls under a case if the instance appears in the model file with a path matching the block path pattern, all the selection parameters present and matching the values specified for the case, and all the transmitted parameters present and satisfying any checks which may be imposed by their translation mode.

It is possible that a library block might fall under more than one case. The case selected will be the *last* matching case in the metadata. The effect of this is that if two methods are provided of translating a simulink block, one of which matches a subset of the instances of that block which are mapped by the other, then the less general case should be specified last in the metafile if it is ever to be invoked. This is relevant, for example, where a block may take either a Scalar or a Vector parameter, but a different Z specification is required in each case. This selection algorithm makes it possible for the user effectively to override a definition in the supplied library, either in special cases or in general, by supplying alternative definitions selected by his own metadata, provided that his metadata is appended to the metadata for the supplied library.

The information for a case in the library metadata file is referred to as a block specification and takes the following format:

```
BlockSpecification {
  ASname              WORD
  Zname               WORD
  HeldZname           WORD
  ResetZname          WORD
  BlockPath           PATTERN
  InputPortTypes      PORTTYPES
  OututPortTypes      PORTTYPES
  UsedLocalVariables  NAMES
  SelectionParameters {
    required block parameters in same format as model file
  }
  TransmittedParameters {
    parameters to be passed with their translation codes
  }
}
```

The ordering of the components of a block specification is not important.

The ASname parameter is optional. It is typically used in metadata that is automatically generated when a library is processed (see section 3.5). When it is supplied, the Zname parameter must identify the name of the Z translation of a block in the artificial subsystem given as the value of the ASname parameter. A block specification with an ASname parameter will only be selected for the translation of a block which has been given a block modifier specification (see section 3.2.3) identifying the same artificial subsystem. A block specification with no ASname parameter will only be selected for the translation of a block which has no block modifier specification.

When a block reference in a model refers to that blockblock modifier specification (see section 3.2.3) has been given for the block reference. In those circumstances, a block specification with an ASname parameter will only be selected if the value of the parameter is the same as the ASname value defined in the block modifier specification.

The Zname parameter must be given. It identifies the target Z specification for this case. The Z specification for the selected case will be invoked using the name given by this parameter, supplying

to it as an argument the transmitted parameters in a binding display using the parameter names as component names (in the case that there are transmitted parameters).

The HeldZname and ResetZname parameters are optional. They should always be supplied if the block in question has internal state. If they are supplied, then when the case is selected and an instance of the block is encountered in an action system or a library, the values of these parameters should identify the Z specifications for the "held" and "reset" behaviour for the block. These specifications will be invoked in exactly the same way as the one identified by Zname.

The BlockPath parameter is optional. The PATTERN supplied for this parameter should be in the same format as those used for output selection (see section 3.2).

The InputPortTypes and OutputPortTypes parameters are optional. They must consist of a sequence of characters with one entry for each input port, in ascending order with no gaps (i.e. no omitted ports) and no spaces, the whole enclosed in double quote marks (`"`). Each entry must be "S", "V" optionally followed by a numeral, "B" followed by an optional numeral and a bus structure definition, "G", or "U" standing for port types *ScalarPT*, *VectorPT*, *BusPT*, *GenericPT*, *UnknownPT* respectively. If the string is shorter than the number of input ports then the unspecified ports are taken as of type *UnknownPT*. The numeral suppled for a vector or bus indicates the width of the signal, its absence or the value "0" indicating that the width is unknown. A bus structure definition consists of a comma separated list of signal types enclosed in square brackets. Each signal type is a (possibly empty) signal name followed by a colon and then a type indication for that signal (as described by this paragraph). Type indications used in bus structures may not be "U" or "G", and if "V" is used it must be followed by a non-zero numeral. The width of a bus need not be specified, and will be computed from the bus structure. If it is specified it will be checked against a computed value and the parameter will be rejected if the width specified does not match that derived from the bus structure. The distinction between "S" and "V" should be based on the type of the port in the Z specification, not on the width of the signal. The "G" case should only be used where the corresponding Z specification is generic in the type of the ports shown as G, when they all have the same generic type (and therefore the same type when instantiated), and when, if this is a vector type, they will also have the same width.

The following is the more formal description of the grammar of the port type parameter taken from the formal specification of ClawZ [2]:

```
<signal_name_a> ::= ([^ ":" "\" ] | "\" [^])*
<signal_type> ::= <signal_name_a> ":" <port_type>
<bus_structure> ::= <signal_type> ("," <signal_type>)*
<port_type> ::= "S" | "V" (<numeral>|)
        | "B" (<numeral>|) "["<bus_structure>"]"
        | "G" | "U"
<port_types> ::= <port_type>*
```

The UsedLocalVariables parameter is optional. If present it should consist of a comma separated list of local variables enclosed in double quote marks. The names are those of variables masked in some enclosing masked subsystem whose values are to be passed from the calling context to the Z specification (in the same binding as the transmitted parameters).

TransmittedParameters should be omitted if the specification is not parameterised.

The supported parameter translation codes are as follows:

Quoted Parameter passed enclosed in quotes, but otherwise unchanged and unchecked.

Unquoted Parameter passed without enclosing quotes, but otherwise unchanged and unchecked.

Scalar Parameter parsed as a scalar Matlab expression complying with a grammar supported by ClawZ and translated into a Z expression.

Vector Parameter parsed as a vector Matlab expression complying with a grammar supported by ClawZ and translated into a Z expression. A scalar expression is accepted and passed as a unit vector.

Matrix Parameter parsed as a Matlab matrix expression complying with a grammar supported by ClawZ and translated into a Z expression. A row vector is accepted. translated and enclose in an extra pair of sequence brackets yielding a unit sequence of sequences. A scalar parameter is accepted and passed as a one-by-one matrix.

Unified Parameter parsed as a Scalar, Vector or Matrix (in that order of preference), translated into Z and converted into a unified Matlab value type using one of the functions "S2U", "V2U", and "M2U".

Fcn Parameter parsed as a Scalar Fcn expression using the grammar described in the Simulink manual entry for the Fcn block, and then translated into Z and wrapped in an abstraction over the variable u. The variable u must be used consistently in the expression either as a vector as a scalar (i.e. either always with a subscript or never with a subscript).

Checkbox A parameter of this type must either have the value "off" or "on" and will be automatically translated into Z as "real 0" or "real 1" respectively.

"Popup(*popuplist*)" A popup translation code consists of the word "Popup" followed by a list of entries for a popup list, separated by "|" characters and enclosed in brackets, the whole translation code must also be enclosed in double quotes. The value supplied for such a parameter must correspond exactly to one of the entries in the list and will be translated into a Z expression of the form "real n", where "n" is the numeric position in the popup list of the entry supplied.

Note that quotation marks must be present for the Popup parameter translation code and must not be present for any other parameter translation code.

A parameter may be present both as a selection parameter and as a transmitted parameter.

B.3.1 Translatable Matlab Expression Syntax

When block parameters of type *Scalar*, *Vector*, *Matrix* or *Unified* are translated by ClawZ a subset of the Matlab expression syntax is accepted. This same language subset is accepted in the right hand side of equations in .m files submitted to ClawZ for translation into Z.

We provide here an informal account of the language subset supported by ClawZ. A fuller and more formal specification of this language may be found in [2].

The language supported may be thought of as *scalar* expressions with some vector and matrix facilities (for use only in parameters of type *Vector*, *Matrix* and *Unified*). The constraints imposed on scalar expressions are similar to those imposed by Simulink on the expressions allowed as parameters to Fcn blocks.

First we describe the scalar expression language, then the ways in which vectors and matrices can be constructed from scalars. The vector and matrix facilities supported can only be used in constructing vectors and matrices as parameter values, not for vector and matrix values which are used inside a scalar expression.

Scalar expressions are formed from literal scalar values (i.e. integer or floating point constants), or from scalar variables which have been defined in .m files or from vector or matrix variables subscripted by one or two scalar expressions respectively. Vector and matrix variables can only be used when subscripted to yield scalar values. No array operations other than subscripting are supported on such variables in scalar expressions. These values may be combined by the usual range of scalar arithmetic operations, and by the relational operators (yielding reals as truth values) and by the boolean operators. In addition, some functions built into Matlab, e.g., the trigonometric functions, are supported. The supported functions are those in the Matlab function library which are documented as acceptable in Fcn expressions.

A vector expression may be formed in the following ways:

- as the name of a variable defined in a .m file
- as a vector display consisting of a sequence of scalars enclosed in square brackets and separated by spaces or commas. The form of the scalars allowable in such a display is restricted. Scalar literals are supported, scalar expressions are also supported (as defined above) but these must not contain any infix operators unless these are in a subexpression (or the whole expression) enclosed in brackets.
- one case of the ":" notation may be used as a vector (but not inside a vector display). This is the case with just two operands in which the operands are either scalar literals or scalar expressions (as defined above).
- the name of a vector valued variable defined in a .m file subscripted by a vector expression (as here defined).

A matrix expression may be formed only as a matrix display in which the separator ";" occurs at least once to create more than one row. The elements allowed as scalars in the display are the same as those allowed in vector displays.

B.3.2 Translatable Fcn Expression Syntax

When parameters of type Fcn are translated expression syntax supported is as described in the Simulink documentation for the Fcn block except that the use of the "u" variable for referring to

inputs must be consistent in the expression, i.e. it must always or never be used with a subscript. Violation of this requirement will not be detected by ClawZ but will result in an error when the output specification is type-checked. To get type correct Z it is also necessary that the usage of "u" match the type of the signal input to the Fcn block.

B.3.3 Dependence of Expression Translation on the ClawZ Library

It should be noted here that the translation of expressions undertaken by ClawZ is a simplistic syntactic transformation, and will only give correct Z if that Z is interpreted in the context of a correctly constructed Z library. In particular, ClawZ assumes that the precedence relations between the various operators in the expression languages is preserved in the corresponding operators in the Z library, and takes care to preserve the relevant bracketing structure in the expression.

The library supplied with ClawZ is written to correspond with the documented precedence of the various Matlab and Fcn operators. Matlab and Fcn functions are specified by fixity and type only. The full extent of the correpondence or divergence between operators and functions in Matlab and Fcn expressions is not clear from the documentation. The ClawZ translator therefore uses distinct names for Matlab and Fcn operators and functions, even where these appear to be the same. In some cases this is necessary because of differences in precedence, in others the distinction is made so that any semantic divergence which might later become clear can be corrected by amendment to the library without requiring changes to the translator.

B.4 Work-arounds for Translator Limitations

It may be noted that the library facilities may be used to overcome limitations in the translator in a systematic and traceable way. For this reason use of these facilities may be considered preferable to manual editing of partial translations where a full translation is not possible.

A typical reason for the translator failing to provide a full translation might be the use of a block which accepts parameters which cannot be translated (e.g. special character strings, or expressions outside the supported language subset).

Because of the flexibility of the facilities it is difficult to anticipate the ways in which it might be used. In extremis the facilities could be used to provide a hand translation of a block which is selected using the blockname in the diagram, or using a pattern for matching against the full hierarchic pathname of the block. More commonly, a block which is normally used with a Matlab expression parameter can be provided with different specific translations for any values of the parameter which cannot be automatically translated into Z by ClawZ.

It is recommended that where the metadata and/or library are extended for purposes specific to a particular model this be done in a separate document which is clearly application specific and where the justification for the various extensions can be given. If the metadata extracted from this document is appended to the library metadata then it will take precedence during the translation, and may be used (if necessary) to supersede definitions already in the library.

This method of overcoming translator limitations will only work for translating blocks which are *transparent* (see [1] for a description of this term). Where a block is opaque (e.g. trigger and enable blocks) it is unlikely that any translation will be correctly composed by the diagram semantics implemented in the ClawZ translator.

C CONTROLLING THE MAPPING OF SIMULINK NAMES

The translation of names and paths from Simlink models into Z specifications is awkward, partly because of the lexical rules for Z identifiers and also because of the need to avoid clashes with names generated by the compliance tool.

ClawZ provides flexibility to the user in how this mapping is undertaken, but this results in some complexity in the user controls. ClawZ takes some steps to ensure that the Z translation of Simulink names results in a unique Z name for each Simulink name. However some of the user controls can subvert the algorithms used and so care should be taken in setting these controls.

Simulink blocks have paths which are essentially a sequence of the block names of all the enclosing subsystems ending with the name of the block itself. The name of the Z schema which translates the block itself must be globally unique, and is therefore obtained by translating the complete path. For some purposes a local name suffices, notably in the signature of the schema defining the subsytem containing a block. These two kinds of name for a block are interlinked by forming the translation of a path from the translation of the blocks in the path.

C.1 Translating Local Names

Because Simulink names may contain characters which are not permitted in Z identifiers a translation which guarantees against introducing ambiguity (and possibly translating two different block names to the same Z identifier) is liable to be either opaque or prolix or both. A mechanism is therefore provided allowing information to be discarded, risking clashes, but using numeric suffixes where necessary to resolve the clashes. Two basic translation methods are available, one using a translation table giving a verbose translation of all characters not legal in Z (e.g. "slash" for "/") the other using a single nominated character or character sequence to replace all the characters not allowed in Z. The user selects which of these methods is adopted by setting the replacement string in the control *z_name_filler*. If it is set to a non-empty string then that string replaces all non alphnumeric characters in the blockname. If the *z_name_filler* is empty a translation table will be used, either a default table or one supplied by the user using the procedure *set_translation_table*.

After a local name is translated using the filler string or the translation table on a character by character basis, the following further transformations are undertaken.

1. Depending upon the selected method of composing local names into global path names some or all instances of a separator character will be removed (see below).
2. Leading and trailing underbar characters are discarded.
3. If the name then begins with a decimal digit, or is empty, a "Z" character is added at the front.
4. If the name clashes with the translation of some local name already translated in the same context (i.e. in the same subsystem of the Simulink model) then a numeric suffix is added, which is the smallest positive integer which yields a translation which has not already been used in this context.

The translation is then remembered in the specific context so that no other names will be given the same translation.

C.2 Translating Global Names

Global names are the translations of complete Simulink paths. The global names are compounded from the local names which translate the names in the path in the specific context of those names. There are two methods of doing this from which the user may select using the *double_separator* flag, and the *z_path_separator* string control. The *z_path_separator* string control should be set to a single character by the user (or left at its default setting of "_").

If the *double_separator* control is set to true, then the translation of the local names will collapse all sequences of the separator character which occur in the translation to one occurrence of the character and will then use two occurrences between each local name in the translation of a path for use as a global name.

If the *double_separator* control is set to false, then the translation of the local names will remove all occurrences of the separator character which occur in the translation and will then use a single instance of the separator character between each local name in the translation of a path for use as a global name.

In compounding the local names to yield a global names any "Z" characters which were added to the local names so that they did not begin with decimal digits are removed, and those used to replace the empty name, except for a Z added to the very first name in the path.

C.3 Name Mapping Controls

In addition to the translator steering file described in section 3.4, there are four string controls and a flag which may be used to alter the mapping of Simulink names into Z (see also section C.6). To change the value of a string control you use the command `set_string_control`, prior to calling `clawz_run` or `mfile_run`. For example, to cause Matlab variable names to be prefixed with an 'X', you would use the command:

```
set_string_control("z_name_prefix", "X");
```

To change the value of a flag you use the command `set_flag`, prior to calling `clawz_run` or `mfile_run`. For example, to set `double_separator` mode:

```
set_flag("double_separator", true);
```

The purpose of the four string controls are described in the following table:

Control Name	Description
`z_name_filler`	By default, punctuation marks and other non-alphanumeric characters in Simulink names are translated into Z using a table of mnemonic names. E.g., ',' is translated into 'Comma'. If you set the string control `z_name_filler` to a non-empty alphanumeric string, then that string will be used in place of the mnemonics as the translation of all non-alphanumeric characters. This string control applies to all Simulink names (block names, subsystem names and Matlab variable names).
`z_path_separator`	This string control should be set to a single character string and defaults to "_". Either one or two instances of this character (according to the setting of the control *double_separator*) are used to separate the local names in the translation of a complete Simulink path into a global Z identifier.
`double_separator`	This flag determines how many instances of the `z_path_separator` character are used to separate the local names in forming a global name. If it is set two are used, otherwise one. It also controls whether instances of the `z_path_separator` are permitted in the translation of the local names. If the control is set to "true" single occurrences are permitted, and multiple occurrences are replaced by a single occurrence. If the control is set to "false" all occurrences of the `z_path_separator` are removed from the translation of a local name.
`z_name_prefix`	This string control gives a prefix which is inserted at the beginning of any Matlab variable name when it is translated into Z (by the .m file translator or by the Model translator when a Matlab or Fcn expression is translated into Z). This does not apply to block names and subsystem names.
`z_name_suffix`	This string control gives a suffix which is appended to any Matlab variable name when it is translated into Z (by the .m file translator or by the Model translator when a Matlab or Fcn expression is translated into Z). This does not apply to block names and subsystem names.

C.4 Setting the Translation Table

To set the translation table you use the command `set_translation_table`, prior to calling `clawz_run` or `mfile_run`. A single parameter is required which must be a list of pairs of strings of which the first string must always be a single character. Characters not mentioned in the mapping will be left unchanged.

E.g.:

```
set_translation_table [
("/", "slash"),
("?", "q")
];
```

The above simple example would not be suitable if there were spaces or newlines or control characters or punctuation characters other than '/' and '?' in the names in the model. To give a more typical example, suppose that inspection of a model showed that with one or two exceptions names were all formed from alphanumeric characters using spaces, newlines and hyphens to separate elements of the names. The following code for setting the translation table would map the names into Z in a natural way, using a single underscore to represent spaces, newlines and hyphens, the string "XXX" to highlight any exceptional characters.

```
fun my_trans ch = (
    if          "a" <= ch andalso ch <= "z"
          orelse "A" <= ch andalso ch <= "Z"
          orelse "0" <= ch andalso ch <= "9"
    then   ch
    else if       ch = "-"
          orelse  ch = " "
          orelse  ch = "\n"
    then   "_"
    else   "XXX"
);

set_translation_table
    (map (fn i => (chr i, my_trans (chr i)))
          (interval 0 255));
```

C.5 The NameMapping table

In addition to all the above controls the user may chose local names on a block-by-block basis by using the `NameMapping` structure in the ClawZ steering file (see section 3.4.1). The Z names supplied by the user in the `NameMapping` structure must comply with the following constraints:

- They must be lexically valid ProofPower-Z words.
- They may not begin or end with the `z_path_separator` character or the underbar character.
- They may not contain more than one consecutive `z_path_separator` character.
- If the `double_separator` flag is false they may not contain any occurrences of the `z_path_separator` character.

If it is required to set the ports types for a subsystem the path of the subsystem itself should be used rather than the path of the port blocks within the subsystem, provided that all the input or all the output port types are to be set. Types for individual input or output ports can be set by setting the the output port on the input port block within the subsystem, or the input port of the output port block within the subsystem respectively.

C.6 Action Subsystem Schema Names

The suffixes used in forming the names of the active, held and reset schemas when an action subsystem is translated are, by default, "$_a$", "$_h$" and "$_r$" respectively. You may change these using string controls called *active_suffix*, *held_suffix* and *reset_suffix* respectively. For example, if you used the commands

```
set_string_control("active_suffix", "_active");
set_string_control("reset_suffix", "_reset");
```

the Z translation for an action subsystem would take the form:

$$subsys\_\_name \mathrel{\widehat=} Active \land subsys\_\_name\_active \lor Inactive \land subsys\_\_name\_reset$$

rather than the default of:

$$subsys\_\_name \mathrel{\widehat=} Active \land subsys\_\_name_a \lor Inactive \land subsys\_\_name_r$$

D CLAWZ RUN PARAMETER TYPE DECLARATIONS

We provide here the ML source of the declarations for the parameter type required for initiating a run of clawz.

```
signature ClawZControl = sig
type OUTPUT_FILTER_SPEC = {
        excl            :       string list,
        incl            :       string list,
        file_name       :       string OPT};

datatype BLOCK_MODIFIER =
                Include of string list
        |       Exclude of string list
        |       ASname of string;

type BLOCK_MODIFIER_SPEC = {
        path            :       string,
        filter          :       BLOCK_MODIFIER};

type ART_SUBSYS_SPEC =  {
        name            :       string,
        top             :       BLOCK_MODIFIER_SPEC,
        rest            :       BLOCK_MODIFIER_SPEC list,
        output_spec     :       OUTPUT_FILTER_SPEC list};

type RUN_PARAMS = {
        meta_file       : string,
        model_file      : string,
        steer_file      : string OPT,
        meta_output     : string OPT,
        output_spec     : OUTPUT_FILTER_SPEC list,
        art_subsys_specs        : ART_SUBSYS_SPEC list};

val clawz_run : RUN_PARAMS -> unit;

type M_FILE_RUN_PARAMS = {
                        parameter_type : string,
                        m_file : string,
                        output_file : string OPT,
                        types_file : string OPT};
val m_file_run : M_FILE_RUN_PARAMS -> unit;
```

```
val diag_a_blocks : (string * ClawZTypes.A_BLOCK) list ref;

val set_translation_table : (string * string) list -> unit;

val default_translation_table : (string * string) list;
end (* of signature ClawZControl *);
```

E OUTPUT FILE ENCODING

The output from ClawZ may be written either in the ProofPower ASCII representation of Z, or in the ProofPower extended character set. The flag "output_extended_characters" controls which encoding is used, and should be set before calling `clawz_run` or `mfile_run` as follows:

```
set_flag("output_extended_chars", bool);
```

where *bool* is either `true` for extended character set or `false` for pure ASCII.

F LENGTH OF LINES IN THE Z OUTPUT FILES

ClawZ attempts to restrict the length of lines in the Z output files it produces to the number of character given by the integer control *z_line_length*. This controls the insertion of line breaks in constructs such as the binding displays that give the parameters to library blocks. The default value of this control is 80 characters. To change it, use the command *set_int_control* as in the following example:

```
set_int_control("z_line_length", 120);
```

G SUMMARY OF FLAGS AND CONTROLS

The following table shows each of the flags, integer control and string controls that affect the operation of ClawZ. Each row in the table gives the name of the flag or control, its default setting and a reference to the section and page of this document where a description may be found.

Flags		
double_separator	false	C.3 page 46
inhibit_output_on_error	true	3.3 page 17
output_extended_chars	false	E page 50
virtualize	false	3.6 page 21
Integer Control		
z_line_length	80	F page 50
String Controls		
active_suffix	"$_a$"	C.6 page 48
held_suffix	"$_h$"	C.6 page 48
porttype_dumpfile	""	3.3 page 17
reset_suffix	"$_r$"	C.6 page 48
z_name_filler	""	C.3 page 46
z_name_prefix	""	C.3 page 46
z_name_suffix	""	C.3 page 46
z_path_separator	"_"	C.3 page 46