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

ClawZ
—
Model Translator Specification

Version:	10.14
Date:	26 January 2004
Reference:	ZED504
Pages:	238
Prepared by:	R.B.Jones
Tel:	+44 1344 642507
E-Mail:	RBJones@RBJones.com

0 DOCUMENT CONTROL

0.1 Contents

0.2 Document Cross References

[1] LEMMA1/DAZ/PLN019. *Control Law Tool — A Proposal to DERA, Malvern.* R.D. Arthan, Lemma 1 Ltd., `rda@lemma-one.com`.

[2] LEMMA1/DAZ/PLN022. *Control Law Project Plan.* R.B. Jones, Lemma 1 Ltd., `rbjones@rbjones.com`.

[3] LEMMA1/DAZ/PLN035. *Reserved for RBJ.* R.B. Jones, Lemma 1 Ltd., `rbjones@rbjones.com`.

[4] LEMMA1/DAZ/USR505. *ClawZ User Guide.* Lemma 1 Ltd., `http://www.lemma-one.com`.

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

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

[7] LEMMA1/DAZ/ZED506. *ClawZ - Z Library Implementation.* R.B. Jones, Lemma 1 Ltd., `rbjones@rbjones.com`.

[8] *Using Simulink, Version 2.2.* The MathWorks Inc., 1998.

0.3 Changes History

Issue 2.1 First draft issue to DERA.

Issue 3.1 First contract phase 1.

Issue 4.2 First contract phase 2.

Issue 5.1 First contract phase 3.

Issue 6.1 Real ClawZ, first issue.

Issue 7.1 Real ClawZ, second issue.

Issue 7.2 Minor upgrades to support the specification of ClaSP.

Issue 7.3-7.18 ClawZ Extensions contract (January 2002).

Issue 8.1 ClawZ Extensions contract extension (JANUARY 2002).
Synthesis.

Issue 9.1- 10.2 ClawZ extensions II contract (JULY 2002).
Virtualization.

Issue 10.2 - 10.7 Action subsystems in ClawZ contract (first stage, FEBRUARY 2003)

Issue 10.8 - 10.9 Action subsystems in ClawZ contract (second stage, MAY 2003)

Issue 10.11 Artificial subsystems of libraries (JUNE 2003)

Issue 10.12 Elimination of output duplication (JULY 2003)

Rewrite of sections 7.2 and 7.3. *output_path_match* remains unchanged, all other functions are changed or new.

Issue 10.13- 10.14 Updates for changes in ProofPower version 2.7.3 (JANUARY 2004).

Z universal set is now called $\mathbb{U}$; "|" is now treated as a punctuation symbol and so cannot be used for the names of the Z constants for Matlab and Fcn operators.

Incorporated new information about the syntax of exponentiation in Fcn expressions.

0.4 Changes Forecast

The following changes have been discussed as possible future extensions to ClawZ.

1 *If* block parameter translation codes.

2 Mask variable name clash elimination.

3 Flexible selection parameter matching.

4 Extension to model file EXPRESSION syntax.

1 GENERAL

1.1 Introduction

This document is one of the deliverables from the Control Law project, placed by DERA Malvern with Lemma1 Ltd. For the relevant proposal see [1] and for the plan see [2]. This specification is produced in the light of the discussion of relevant semantic issues in [5].

The following Standard ML script creates a new theory for this specification.

SML
```
open_theory "z_library";
delete_theory "zed504" handle _ => ();
new_theory "zed504";
new_parent "cn";
```

1.2 Notation and Conventions

The specification is primarily presented in ProofPower-Z, together with appropriate annotation in plain English. We include here the definition of the conditional clause, which is not available in the ProofPower-Z libraries.

Z
```
fun if _ then _ else _
```

Z
```
[X]
    if _ then _ else _   : (BOOL × X × X) → X
─────────────
    ∀ b:BOOL; x,y:X •
           (b     ⇒ (if b then x else y) = x)
    ∧      (¬ b   ⇒ (if b then x else y) = y)
```

The following definition is borrowed from the compliance notation specifications:

$OPT[X]$ denates an optional member of X. To represent this, $OPT[X]$ comprises all subsets of X with at most one element:

Z
```
OPT[X] ≙ {A : ℙX | ∀x, y: A• x = y}
```

z

$[X]$

$Value : X \rightarrow OPT[X];$
$Nil : OPT[X]$

$\forall x \colon X \bullet\ Value\ x = \{x\}\ ;$
$Nil = \{\}$

The following is borrowed from the ClawZ library (which is in general incompatible with this specification). It is an integer variant of the similarly named matlab operator (defined below).

z

$fun\ 2\ leftassoc\ \_ :_z \_$

z

$\_ :_z \_ : (\mathbb{Z} \times \mathbb{Z}) \rightarrow seq\ \mathbb{Z}$

```
∀x, y: ℤ•
        x :z y =
        {i,z :ℤ
        |       i = z − x + 1
        ∧       x ≤ z ≤ y}
```

1.3 Overview

The translator will translate two different kinds of document into Z.

1. Simulink .mdl files (either *Models* or *Libraries*).
2. Matlab .m files.

There are five sources of information which are required by the translator:

1. The file to be translated.
2. A steering file
3. A library metadata file (not required for .m files).
4. Parameters to the run of ClawZ
5. Various string controls and flags which may be used to modify the operation of clawz.

The model file provides the details of the model to be translated into Z in the form of wiring diagrams which combine Simulink library blocks into a number of subsystems and a single main system.

In order to process a Simulink model after translation into Z there must first be made available specifications in Z of the library blocks which are used in the model. This will be in the form of one or more ProofPower-Z theories.

In order to make use of these libraries the translator will need some "metadata" providing information about the mapping from instances of Simulink library blocks to Z specifications. This is the content of the "library metadata" file, which defines the mapping from Simulink blocks to Z specifications. Information about the types of signals associated with the different library blocks is also provided in this file.

Two distinct strategies are available for mapping Simulink/Matlab values into Z values. In the first different types in Z are used for scalars, vectors and matrices, a vector being represented as a sequence of scalars and a matrix by a sequence of vectors. This gives rise to some problems in translation since the translator is not always able to determine the correct type to use. The translator also supports translation to Z using a single type for all values. Block parameters are still mapped into expressions which are formed using sequence displays, but these are then mapped into the unified value space using appropriate injection functions.

Control over the mapping strategy is exercised through the metadata, which specifies the translation mode for block parameters. Where distinct types are require the parameter types Scalar, Vector and Matrix are used as appropriate. Where a single type is require the parameter type Unified should be used.

Control over .m file translation involves a similar choice, but the single parameter type SVM is used to indicate that the .m file translation should map expressions into distinct Z types according to their structure.

Further information on the construction of libraries and their metadata may be found in the ClawZ user guide [4].

This specification proceeds by defining abstract data structures corresponding to the information provided in these files, including the resulting output files, and then defining the mapping from the abstract representation of models and metadata to that of Z specifications.

2 SYNTACTIC STRUCTURES

In this section we specify the concrete syntax of the various files which are used or produced by the translator and the library extractor.

The files relevant to ClawZ are as follows:

1. Simulink model files

2. Simulink .m files
3. Library metadata files
4. ClawZ steering files
5. Z specification files
6. Standard output from ClawZ
7. Matlab variable type information

Of these, items 1, 3 and 4 use a common structure, referred to informally as model file format.

2.1 Model File Lexis

No explicit account of the Lexis is available to us. On the basis of the available examples the following is inferred.

For the purposes of ClawZ files in model file format are considered to have a two level lexis. The outer level is described here, and determines a limited level of lexical analysis which is undertaken as files in this format are read into ClawZ. Within this structure there occur quoted strings which may have significant internal structure and may be subject further lexical and syntactic analysis after the file has been read in.

Files consist of 8-bit ascii characters. The characters are grouped as follows:

whitespace space, tab, carriage return, linefeed

delimiters Double-quote (`"`), curly braces

alpha upper or lower case alphabetic

digit 0-9

special characters Any other ASCII character.

The lexical tokens are:

SPACE a non-empty sequence of whitespace characters

OPEN the open curly brace

CLOSE the close curly brace

NAME A sequence of characters terminated by (unescaped) white space or a double quote character ("`"`"). Backslash may be used to escape characters which would otherwise terminate the "name", and must therefore also be used to escape backslash characters (i.e. two backslash characters must be supplied for every backslash desired in the name). A single backslash characters will always be discarded even if the character it precedes need not have been escaped. Backslash must also be used before the first character if that character would be legal as the start of any other lexical token, i.e. if it is: white space, open of close brace, open or close square bracket, digit or double quotes. Newline carriage return or tab characters may be entered as "`\n`", "`\r`", and "`\t`" respectively.

-

This liberal lexis is intended to allow Simulink paths to be accepted in those places where the syntax calls for a *NAME*.

NUMBER an lscalar as specified in the section on parameter translation (4.6).

EXPRESSION An expression beginning with an open square bracket and terminating with a matching closing square bracket.

QVALUE a sequence of characters enclosed in double-quotes, possibly interrupted by linebreaks which must be preceded and followed by double-quote characters. Double-quotes are not permitted, unless preceded by a backslash in which case they do not terminate the quotation. Backslash characters when not used for escaping must also be escaped by backslash. Lexical processing should discard the escape characters leaving only the characters escaped, this applies even when a character is unnecessarily escaped, i.e. all single occurrences of backslash are discarded, as well as alternate backslashes in any sequence of more than one backslash the character immediately following any discarded bacslash is always included but its value ignored when considering any special action (like termination or escaping).

In the sequel the internal structure of various kinds of QVALUE is discussed and specified. In all the grammars supplied for this purpose it is assumed that the first level of escape processing required to lex a QVALUE has been undertaken before parsing according to the specified grammar. Conversely, where pretty printing of values for output as QVALUEs is specified, a final stage of supplementary escaping is required as the expression is made into a QVALUE before output.

COMMENT any line beginning with a hash ("**#**") is a comment and is ignored by the translator

2.2 Model File Syntax

The concrete syntax is exemplified in Annex B of [8].

The following notes supplement that Annex.

A context free grammar of the file format to be supported is as follows:

```
<value> ::=
              NAME
      |       NUMBER
      |       QVALUE
      |       EXPRESSION
      |       OPEN <parlist> CLOSE

<parlist> ::= (NAME <value>)*
```

A model file is a nonempty "parlist" of which the first NAME is "Model" or "Library" and all subsequent NAME/value pairs are ignored by the translator. Other files such as the metadata and steering files which use the same format are not subject to this constraint.

The use of this grammar depends upon the assumption that the value of a parameter never begins with an OPEN (curly brace) unless it is a struct, and that a simple grammar can be established for *EXPRESSION* permitting this to be treated as part of the Lexis.

2.3 .m File Lexis and Syntax

A .m file consists of a sequence of logical lines, each of which is formed from one or more physical lines and contains zero or more matlab commands.

A physical line is terminated by a carriage return (`\r`, used under UNIX), a line feed (`\n`, used under MACOS) or by carriage return then line feed (`\r\n`, used under DOS/Windows). CR/LF will be treated as a single physical line terminator. Any of these will be accepted irrespective of the system under which ClawZ is running, and will be referred to below as a "newline".

A logical line begins at the beginning of the file or after the end of the previous logical line, and ends at the first newline character which is not either:

1. inside the brackets of a matrix display
2. preceded by a line continuation (after the previous newline character).

The specifications of the syntax acceptable in .m files are written as if:

1. newlines in Matrix displays are replaced by semicolons
2. elided newlines and comments are discarded

Comments are allowed starting at any point of a physical line with a "%" character and proceeding to, but not including, the next newline character (the end of the physical line). Line continuations are permitted and are indicated by the sequence "..." occurring as the last non-space non-comment characters on the physical line. Such a continuation marker (the three dots and all characters up to and including the end of line) is treated as a space. A non-elided newline may only appear within a logical line inside the braces of a matrix display, (in which case it is treated as a semi-colon, terminating a row) and will otherwise terminate the logical line.

A logical line consists of zero or more matlab commands separated by commas or semicolons. A restricted subset of matlab commands can be translated into Z. Translatable commands are equations with a Matlab name on the left and an "allowable" Matlab expression on the right. Details of the structure of Matlab names and allowable Matlab expressions are give in section 3.7.2 on parameter translation.

The translator will tolerate non-comment commands which are not equations, but these commands will not be translated and will cause all subsequent commands on the same logical line to be ignored. The same effect will result if an equation is invalid or falls outside the range of "allowable" expressions supported by the translator.

Matlab names are significant only in the first 31 characters.

2.4 Model Expression Syntaxes

Two kinds of expression occur in Simulink models as parameters to Simulink blocks. The analysis of these expressions is specified and implemented as a second layer of analysis, taking place subsequent to the initial analysis of the model file.

The two kinds of expressions are:

1. Fcn block expressions
2. Matlab expressions

The first occur only as parameters to *Fcn* blocks, while the second appear as parameters to many other Simulink library blocks, or as parameters to masked subsystems (not currently supported by ClawZ).

In the following specification of the concrete syntax of these expressions, no clear separation is made between lexical and syntactic matters.

2.4.1 Metanotation

In specifying the parsing and translation of parameters no separation is made between lexis and syntax.

The significant object language characters are: " " (*space*), "[" , "]", ";", ",", ".", "+", "-", "_" the decimal digits and upper and lower case letters.

The metanotation encloses phrase names in angled brackets uses "::=" for defining phrases "|" for choice, round brackets for grouping and postfix "*" for iteration "+" for iteration at least once. The metanotation "[^ *x y z*]" is used to indicate "any character except *x y*, or *z*". The metanotation ""*sc*" – "*ec*"" where "sc" and "ec" are single characters is used to indicate any character with a ascii code between that of sc and that of ec inclusive.

2.4.2 Common Elements

```
<digit>    ::=  "0" – "9"
<letter>   ::=  "a"–"z" | "A"–"Z"
<mlname>   ::=  <letter>(<letter>|<digit>|"_")*
<numeral>  ::=  <digit>+
<lscalar>  ::=  " "* ("–"|)
                (<numeral> ("." (<numeral> |)|) | "." <numeral>)
                (("E"|"e")("+"|"–"|)<numeral>|)
<fname>    ::=  "abs" | "acos" | "asin" | "atan" | "ceil" | "cos" | "cosh" | "exp"
                | "fabs" | "floor" | "hypot" | "log" | "log10"
                | "sin" | "sinh" | "sqrt" | "tan" | "tanh"
<fname2>   ::=  "atan2" | "power" | "rem"
```

2.4.3 Fcn Expressions

These expressions occur in model files as parameters to *Fcn* blocks.

They are essentially scalar expressions in which references to vectors and arrays are only permitted with the necessary subscripts to yield a scalar. One exception is the special vector "u" which is used to refer to the vector of signals input to the *Fcn* block. This is permitted in two non-standard forms, firstly without subscript, to refer to the first element only, and secondly with square brackets instead of round brackets round the subscript (which is not documented, but is frequently used). Except for the usage with square brackets the clause referring to "u" in the following syntax is redundant, since translation of references to "u" will not differ from that of other array references. An intended consequence of this is that when "u" is used without a subscript it will be construed as a scalar, and if it is used both with and without a type error will occur. This allows the translation of *Fcn* blocks with scalar or vector inputs, provided that the use of "u" in the expression corresponds to whether the input is a scalar or a vector. Note however that the scalar/vector distinction here is that between a real and a sequence of reals in the Z output, and not whether the signal is of width 1 or greater.

<f_infix_op>	::=	"+" | "−" | "∗" | "/" | "==" | "!=" | ">" | ">="
		| "<" | "<=" | "&&" | "||" | "^"
<f_prefix_op>	::=	"+" | "−" | "!"
<f_fname>	::=	*<fname>* | "*ln*" | "*sgn*"
<f_at_exp>	::=	*<lscalar>*
		| "(" *<f_exp>* " "∗ ")"
		| *<f_prefix_op>* " "∗ *<f_at_exp>*
		| "*u*" (" "∗ "(" *<f_exp>* " "∗ ")" |)
		| "*u*" " "∗ "[" *<f_exp>* " "∗ "]"
		| *<f_fname>* " "∗ "(" *<f_exp>* " "∗ ")"
		| *<fname2>* " "∗ "(" *<f_exp>* " "∗ "," *<f_exp>* " "∗ ")"
		| *<mlname>* (" "∗ "(" *<f_exp>* (" "∗ "," *<f_exp>* |) " "∗ ")" |)
<f_exp>	::=	" "∗ *<f_at_exp>* (" "∗ *<f_infix_op>* " "∗ *<f_at_exp>*)∗

Note: the Simulink documentation has "pow" rather than "^", but we are assured by the Mathworks that the latter is intended and that "pow" is effectively a synonym for "power".

2.4.4 Matlab Expressions

These may occur as parameters to a wide range of Simulink blocks. The expressions supported are essentially *scalar* expressions, however, selecting elements from arrays is supported as is taking a slice from an array. Array and vector displays are supported at the top level.

Though the following specification combined lexis and syntax, in the case of expressions in .m files there is some very simple lexical preprocessing involved (see 2.3) which is assumed to have taken place. This affects the interpretation of line terminators.

```
<m_infix_op>     ::=    "+" | "−" | "*" | "/" | "\" | ".+" | ".−" | ".*" | "./" | ".\"
                      | "==" | "~=" | ">" | ">=" | "<" | "<=" | "&" | "|" | "^" | ".^"
<m_prefix_op>    ::=    "+" | "−" | "~"
<m_at_exp>       ::=    <lscalar>
                      | "(" <m_sexp> " "* ")"
                      | <m_prefix_op> " "* <m_at_exp>
                      | <fname> " "* "(" <m_sexp> " "* ")"
                      | <fname2> " "* "(" <m_sexp> " "* "," <m_sexp> " "* ")"
                      | <mlname> (" "* "(" <m_sexp> ("," <m_sexp> |) " "* ")" |)
<m_sexp>   ::=        " "* <m_at_exp> (" "* <m_infix_op> " "* <m_at_exp>)*
<row>      ::=        <m_at_exp> (" "* (" " | ",") " "* <m_at_exp>)* " "* (","|)
<rows>     ::=        (<row>? ";")+ <row>?
<vector>   ::=         " "* "[" <row>? "]"
                      | <m_sexp> " "* ":" <m_sexp>
                      | <mlname> " "* "(" <vector> " "* ")"
<matrix>   ::=         " "* "[" <rows> " "* "]"
<parameter>      ::=    <mlname> | <m_sexp> | <vector> | <matrix>
```

Note that a single *mlname* satisfies the grammar for *m_sexp*, but should always be delivered as a *ParName* (see definition of *PARAMETER* in section 3.7.2).

2.5 Mask Parameter Grammars

The following parameters to masked subsystems need to be parsed: *MaskStyleString*, which determines the interpretation of the information in the *MaskValueString*, which gives the actual values for the mask variables, whose names and positions are given in the *MaskVariables* parameter.

```
<popup_name>      ::= ([^ "|" ")" "\" ] | "\" [^])+
<popup_names>     ::= <popup_name> ("|" <popup_name>)*
<mask_style>      ::= ("edit" | "checkbox" | "popup(" <popup_names> ")")
<mask_style_par>::= <mask_style> ("," <mask_style>)*
<mask_value>      ::= ([^ "|" "\" ] | "\" [^])+
<mask_value_par>         ::= <mask_value> ("|" <mask_value>)*
<mask_var>        ::= <mlname> "=@" <numeral>
<mask_var_par>    ::= <mask_var>  (";" <mask_var>)*
```

2.6 Parameter Translation Code Grammar

```
<translation_code> ::= "Quoted" | "Unquoted" | "Scalar" | "Vector"
                     | "Matrix" | "SVM" | "Unified" | "Fcn" | "Checkbox"
                     | "Popup(" <popup_names> ")"
```

2.7 Various Parameter Syntaxes

The grammar for “InputPortTypes” and “OutputPortTypes” parameters in library metadata files (see also section 3.7) is as follows:

```
<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>*
```

Note that in a *signal_name_a*, colon or backslash characters *must* be escaped by backslash, any other character *may* be escaped by backslash. All backslash characters except those escaped by backslash are discarded on input, and a sufficiency of escapes are added on output. A second layer of escaping is added when the *port_types* is quoted. This happens when the information is written to a metadata file but not when it is written to a port type dump file. If a numeral is supplied for a `"B"` *port_type* then its value must be consistent with any detail provided for the *bus_structure*.

The grammar for the “UsedLocalVariables” parameters in metadata files is:

```
<used_local_vars> ::= (<mlname> ("," <mlname>)*)?
```

The grammar for “Ports” parameters in .mdl files is as follows:

```
<ports_param> ::= "[" (<numeral> (" "* (","|" ") " "* <numeral>)*)? "]"
```

The grammar for “OutputSignals” parameters on *BusSelector* blocks in .mdl files is as follows:

```
<signal_name> ::= [^ "." "," ]+
<signal_selection> ::= <signal_name> ("." <signal_name>)*
<outputsignals_param> ::= <signal_selection> ("," <signal_selection>)*
```

The grammar for “Outputs” parameters on *DeMux* blocks in .mdl files is as follows:

```
<outputs_param> ::= <numeral>
                  |       "["(<numeral>(" "* (","|" ") " "* <numeral>)*)?"]"
```

2.8 Block Paths

For the purpose of block selection or output selection it is possible to specify patterns which are compared with the hierarchical block name formed by concatenating all the system and subsystem names and the block name of a particular block.

The concrete representation of such patterns is as follows:

```
|<name>     ::= ([^ "?" "*" "/" "\"] | "\" [^])+
|<pat_el>   ::= "?" | "*" | <name>
|<pattern>  ::= <pat_el> ("/" <pat_el>)*
```

In conversion to its internal form escape characters and "/" separators are removed. A further layer of escaping is implemented by the ML compiler where these patterns are rendered as ML strings, as a consequence of which the "\" characters used for escapes in the above format will need to be doubled up in patterns presented as ML strings.

Similar effects may be observed if it is necessary to enclose a pattern in a metadata file in quotes.

2.9 Steering File Syntax

A ClawZ steering file is similar in syntactic structure to a Simulink model file, consisting of a sequence of name value associations. At the top level of the steering file all the values are structures (i.e. name/value lists enclosed in curly brackets), and the names of these structures must be one of the following:

NameMapping For controlling the Simulink to Z name mapping.

In this case the names are Z identifiers to be used for local names and the values are the quoted full Simulink path of the block for which the Z identifier should be used.

InPortTypes For setting block input port types.

In this case the name should be the full Simulink path of the block whose inport types are being supplied, and the value is a quoted sequence of port types (a < *port_types* >) supplying the types of the input ports.

OutPortTypes For setting block output port types.

Format similar to that for *InPortTypes*.

VariableTypes For information about the types of matlab variables.

In these structures each name is a Z identifier into which a matlab variable name has been translated by the .m file translator, and the value is a (possibly empty) comma separated list of non-negative integer dimensions for the value assigned to the variable enclosed in square brackets ("[]" for scalar values). Some of these dimensions may be zero, which indicates that the .m file translator was unable to make a determination.

In the *InPortTypes* or *OutPortTypes* structures the the port types may be split over multiple lines by supplying additional quoted strings on subsequent lines (as may be done in Simulink model files), e.g.:

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

It should be noted that all the kinds of information permitted in a steering file may be spread over multiple top level structures, and that the effect should be exactly the same as if the content of all the structures of a particular name were concatenated in a single structure of that name. Thus, for example, if supplying matlab variable information from the translation of several different .m files it suffices to concatenate the files from each .m file translator run (with each other and with any other data required in the steering file).

2.10 File Types

This same file format is used both for Simulink models, for library metadata files, and for translator steering files. For the way in which this file format is used in each case see appendices A and B of [8], [6] and [4].

The Matlab .m files have a different format, described above.

Certain other file types have also been mooted in this document. It is not proposed to make use of these other file types in the current version of ClawZ but they are mentioned here nonetheless.

Model Files This is informally described in appendices A and B of Using Simulink [8]. The translator will take no account of any of the information in a model prior to the SYSTEM block which describes the technical content. In particular it is assumed that block parameter defaults are used only for cosmetic features and not to set defaults for parameters which are relevant to the translation.

Library Metadata Files The structure of these is described in [4], which includes an example for use in the integration tests.

Translator Steering Files As described in section 2.9.

3 DATA STRUCTURES

3.1 Structured Files

The abstract outer structure of the Simulink model file and the metadata and translator steering files is as follows.

The information in models is predominantly encoded in parameter name/value pairs. The following given sets are introduced as the sets of possible parameter names and simple values.

z

$$[\boldsymbol{PNAME}, \boldsymbol{PVALUE}]$$

Simple parameter values have a lexical type.

z

$$\boldsymbol{PTYPE} ::= \boldsymbol{SName} \mid \boldsymbol{Number} \mid \boldsymbol{Qvalue} \mid \boldsymbol{Expression}$$

However, a parameter may take a structured value, itself formed from name/value pairs as follows:

z

$$\begin{array}{l} \boldsymbol{STRUCTURE} ::= \boldsymbol{Structure}\ (seq\ [\boldsymbol{name}: PNAME; \boldsymbol{value}: VALUE]) \\ \& \\ \boldsymbol{VALUE} ::= \boldsymbol{Simple}\ (PTYPE \times PVALUE) \mid \boldsymbol{Struct}\ STRUCTURE \end{array}$$

This may be used as an intermediate abstract representation, with a file specific abstract structure used in the main specification as documented below.

3.2 Simulink Models

The following functions coerce between *PVALUE*s and *PNAME*s and character sequences.

z

$$\begin{array}{l} \boldsymbol{pn2sc}: PNAME \to seq\ CHAR; \\ \boldsymbol{pv2sc}: PVALUE \to seq\ CHAR; \\ \boldsymbol{sc2pn}: seq\ CHAR \to PNAME; \\ \boldsymbol{sc2pv}: seq\ CHAR \to PVALUE \\ \hline sc2pn \fatsemi pn2sc = sc2pv \fatsemi pv2sc = id(seq\ CHAR) \end{array}$$

We also require a function for converting a natural number to the *PVALUE* which is its numeral.

z

$$\boldsymbol{num2pvalue} : \mathbb{N} \to PVALUE$$

Certain specific examples of these are given names.

ActionQ is the parameter value “Action?”.

ActionPort is the parameter value “ActionPort” which is the *BlockType* of an action port.

AllowUnequalInputPortWidths is the parameter name “AllowUnequalInputPortWidths”, which is used for Merge blocks.

BlockType is the parameter name “BlockType”, which is used to record the type of a block in a Simulink model.

BusCreator is the parameter value “BusCreator”, which is the *BlockType* of a Simulink bus creator block.

BusSelector is the parameter value “BusSelector”, which is the *BlockType* of a Simulink bus selector block.

Constant is the parameter value “Constant”, which is the *BlockType* of a Simulink constant block.

Demux is the parameter value “Demux”, which is the block type of Demux blocks.

Elements is the parameter name “Elements”, which is the name of a parameter required by a block of type Selector.

ElementSrc is the parameter name “ElementSrc”, which is the name of a parameter required by a block of type Selector.

EnablePort is the parameter value “EnablePort” which is the *BlockType* of a Simulink enable port block.

EnableQ is the parameter value “Enable?”.

held is the parameter value “held” which is used for the *InitializeStates* parameter of the *Action* block.

If is the parameter value “If” which is a block type.

ifaction is the parameter value “ifaction” which is the name of the action port on an *If* action subsystem.

InitializeStates is the parameter name “InitializeStates” which is a parameter to the action block.

InitialOutput is the parameter name “InitialOutput” which is a parameter to the *Merge* block.

initial_state is the parameter value “initial_state” which is used internally for *Merge* block synthesis.

InPort is the parameter value “InPort” which is the *BlockType* of a Simulink input port block.

InputPortTypes is the parameter name “InputPortTypes”, which is used in a library metadata entry to record the types of the input ports.

Inputs is the parameter name “Inputs”, which is used in Merge blocks.

InputType is the parameter name “InputType”, which is used in Selector blocks.

Internal is the parameter value “Internal”, which is used in Selector blocks.

Merge is the parameter value "Merge" which is a block type.

Mux is the parameter value "Mux", which is the block type of Mux blocks.

Name is the parameter name "Name", which is used to record the name of a block or a system in a Simulink model.

NullString is the parameter value "" which is used to check whether a line has a name.

off is the parameter value "off". This is used generally as the value of a parameter selected by a checkbox.

on is the parameter value "on". This is used generally as the value of a parameter selected by a checkbox.

One is the parameter value "1". This is the name of the single input or output port on library blocks of type *OutPort* or *InPort* respectively.

OutPort is the parameter value "OutPort" which is the *BlockType* of a Simulink ouput port block.

OutputPortTypes is the parameter name "OutputPortTypes", which is used in a library metadata entry to record the types of the output ports.

OutputSignals is the parameter name "OutputSignals", which is used in a *BusSelector* block to specify the values on the output signals.

Port is the parameter name "Port", which is used to record the external port number represented by a Simulink port block. This is needed to correctly interpret connections to external ports.

Ports is the parameter name "Ports", which is used to record the number and kind of ports on a block. This is needed to correctly interpret connections to external ports.

Reference is the parameter value "Reference" which is the *BlockType* of a Simulink library reference block.

reset is the parameter value "reset" which is used for the *InitializeStates* parameter of the *Action* block.

Selector is the parameter value "Selector" which is the *BlockType* of a Simulink selector block.

CaseShowDefault is the parameter name "CaseShowDefault" which is a parameter of a Simulink *SwitchCase* block.

ShowElse is the parameter name "ShowElse" which is a parameter of a Simulink *If* block.

state is the parameter value "state" which is used internally for Merge block synthesis.

stateP is the parameter value "state'" which is used internally for Merge block synthesis.

SwitchCase is the parameter value "SwitchCase" which is a block type.

Terminator is the parameter value "Terminator" which is the *BlockType* of a Simulink terminator block.

TriggerPort is the parameter value "TriggerPort" which is the *BlockType* of a Simulink trigger port block.

TriggerQ is the parameter value "Trigger?".

UnitDelay is the parameter value "UnitDelay", which is the *BlockType* of a Simulink unit delay block.

ValuePN is the parameter name "Value", required by the Simulink Constant block.

Vector is the parameter value "Vector" for the Simulink Selector block "InputType" parameter.

X0 is the parameter name "X0", required by the Simulink Unit Delay block.

z

$$\begin{array}{l}
\boldsymbol{AllowUnequalInputPortWidths}, \boldsymbol{Name}, \boldsymbol{BlockType},\\
\boldsymbol{CaseShowDefault}, \boldsymbol{Elements}, \boldsymbol{ElementSrc}, \boldsymbol{Port},\\
\boldsymbol{Ports}, \boldsymbol{InputPortTypes}, \boldsymbol{InitializeStates}, \boldsymbol{InitialOutput},\\
\boldsymbol{Inputs}, \boldsymbol{InputType}, \boldsymbol{OutputPortTypes}, \boldsymbol{OutputSignals},\\
\boldsymbol{ShowElse}, \boldsymbol{ValuePN}, \boldsymbol{X0}: PNAME;\\
\\
\boldsymbol{ActionPort}, \boldsymbol{ActionQ}, \boldsymbol{Constant}, \boldsymbol{BusCreator}, \boldsymbol{BusSelector}, \boldsymbol{Demux},\\
\boldsymbol{EnablePort}, \boldsymbol{EnableQ}, \boldsymbol{held}, \boldsymbol{If}, \boldsymbol{ifaction}, \boldsymbol{initial\_state}, \boldsymbol{InPort},\\
\boldsymbol{Internal}, \boldsymbol{Merge}, \boldsymbol{Mux}, \boldsymbol{NullString}, \boldsymbol{off}, \boldsymbol{on}, \boldsymbol{One}, \boldsymbol{OutPort},\\
\boldsymbol{Reference}, \boldsymbol{reset}, \boldsymbol{Selector}, \boldsymbol{state}, \boldsymbol{stateP}, \boldsymbol{SwitchCase},\\
\boldsymbol{Terminator}, \boldsymbol{TriggerPort}, \boldsymbol{TriggerQ}, \boldsymbol{UnitDelay}: PVALUE
\end{array}$$

z

$$\boldsymbol{port\_block\_types} \mathrel{\widehat=} \{ActionPort, EnablePort, InPort, OutPort, TriggerPort\}$$

Parameters are then represented by schema products:

z

PARAM

$$\boldsymbol{name}: PNAME; \boldsymbol{value}: PVALUE$$

We next introduce some further schema types which are more convenient representations of some of the information which is encoded by parameters in the model.

The *Port* parameter encodes certain information about the numbers and kinds of ports on a block. An abstract encoding of this information is:

z

PORT_PARAM

$$\boldsymbol{max\_in}, \boldsymbol{max\_out}, \boldsymbol{trigger}, \boldsymbol{enable}, \boldsymbol{state}: \mathbb{N}$$

The *OutputSignals* parameter encodes information about the outputs from a *BusSelector* block. An abstract encoding of this information is:

z

$$\boldsymbol{OUTPUTSIGNALS\_PARAM} \mathrel{\widehat=} seq\ (seq\ PVALUE)$$

The *Outputs* parameter encodes information about the outputs from a *Demux* block. An abstract encoding of this information is:

z

$$\boldsymbol{OUTPUTS\_PARAM} ::= OPScalar\ \mathbb{N} \mid OPVector\ (seq\ \mathbb{N})$$

The following information used is used to identify a port, notably in specification of a line.

z

PORT

$$\boldsymbol{block}, \boldsymbol{port}: PVALUE$$

And next a line, connecting a single source port with zero or more destination ports. A line with no destinations may occur in a Simulink model. Lines may have names; the names are not used in ClawZ but are used in ClaSP.

z

LINE

$$\boldsymbol{source}: PORT; \boldsymbol{destinations}: \mathbb{F}_1\ PORT; \boldsymbol{name} : PVALUE$$

Note here that a Simulink model may contain lines without a source port or without a destination port. In both cases these should be silently discarded without causing the translation to be aborted. This is a requirement on the parser and therefore does not appear in the formal description of the translation which begins with the results of the parsing phase. Our grammar for model files also admits the possibility of a line with more than one source port, though we do not believe that this is ever done by Simulink. The translator considers lines with multiple sources semantically ill-formed. Though this does happen in digital hardware it does not appear to be possible in Simulink and our available descriptions of Simulink semantics do not cater for it. The translator will therefore reject models in which lines are found with multiple sources.

We now use a pair of mutually recursive free type definitions to represent systems and blocks. A system is the result of connecting together a number of blocks in a Simulink diagram. A block can either be a Simulink library block, or a subsystem.

z

$$\begin{array}{lll}
\boldsymbol{SYSTEM} & ::= & \boldsymbol{System}\ [\boldsymbol{pars}: \mathbb{F}\ PARAM; \boldsymbol{blocks}: \mathbb{F}_1\ BLOCK; \boldsymbol{lines}: \mathbb{F}_1\ LINE] \\
\& & & \\
\boldsymbol{BLOCK} & ::= & \boldsymbol{SubSystem}\ [\boldsymbol{pars}: \mathbb{F}\ PARAM; \boldsymbol{system}: SYSTEM] \\
 & \mid & \boldsymbol{LibBlock}\ (\mathbb{F}_1\ PARAM)
\end{array}$$

A Simulink model is, for present purposes, just a system.

3.3 Matlab .m Files

A Matlab .m file, for the purposes of this specification is regarded as a sequence of name/value equations (*MFEQ*s).

z

MFEQ

$\textbf{name}: PVALUE; \textbf{value}: PVALUE$

z

$\textbf{M\_FILE} \mathrel{\widehat{=}} seq\ MFEQ$

It should be noted that the names acceptable to Matlab are more restricted than those acceptable in Simulink models, and that there is therefore no need to translate these names to obtain legal Z identifiers. However, a facility for the user to select a systematic transformation of these names is available and is specified as the function *change_mlname*.

3.4 Parameter Expressions

A second level of analysis is undertaken on some of the parameter values (PVALUEs).

This involves analysing them as expressions using the grammar defined in section 2.4.

In this section the abstract structure resulting from this further analysis is modelled.

3.4.1 Common Elements

A sign (or the absence of one) is represented as follows:

z

$$\begin{aligned} \textbf{SIGN} ::= &\ \textbf{Positive} \\ |\ &\ \textbf{Negative} \end{aligned}$$

A floating point literal comes as a sign, a decimal mantissa (value) and a (signed) exponent (power of ten). We model this as a pair of SIGNs and a pair of PVALUEs. The first PVALUE is the decimal value with the decimal point removed. The second PVALUE is the natural number obtained by adjusting the power of 10 following the “e” to take account of removal of the decimal point.

z

PLV

$sign, psign: SIGN; value, power: PVALUE$

(think of PLV as Parameter Literal Value)

3.4.2 Fcn and Matlab Expressions

A common abstract representation is used for these two distinct kinds of expression.

z

$$
\begin{array}{lll}
\boldsymbol{AT\_EXP} ::= & & \\
 & & \boldsymbol{Lscalar}\ PLV \\
 & | & \boldsymbol{Brackets}\ SEXP \\
 & | & \boldsymbol{PrefixOp}\ (PVALUE \times AT\_EXP) \\
 & | & \boldsymbol{Function}\ (PVALUE \times SEXP) \\
 & | & \boldsymbol{Function2}\ (PVALUE \times SEXP \times SEXP) \\
 & | & \boldsymbol{Mlname}\ (PVALUE \times seq\ SEXP) \\
\&
\end{array}
$$

$$\boldsymbol{SEXP} ::= \boldsymbol{Sexpres}\ (AT\_EXP \times seq\ (PVALUE \times AT\_EXP))$$

z

$$
\begin{array}{lll}
\boldsymbol{VEXP} ::= & & \\
 & & \boldsymbol{LvDisplay}\ (seq\ SEXP) \\
 & | & \boldsymbol{LvSlice}\ (SEXP \times SEXP) \\
 & | & \boldsymbol{LvArray}\ (PVALUE \times VEXP)
\end{array}
$$

z

$$\boldsymbol{MEXP} \mathrel{\widehat{=}} seq\ VEXP$$

3.5 Fcn and Matlab Operators

The following declarations indicate the target Z names for the operators which may occur in Fcn or Matlab expressions.

The following are operations over real scalars:

z

$$.+_m,\ .-_m,\ .*_m,\ ./_m,\ .\backslash_m\ : PVALUE$$

The following relations over reals (returning reals) are used as target for the corresponding operators in matlab expressions:

z

$$==_m,\ \sim=_m,\ <_m,\ >_m,\ >=_m,\ <=_m\ : PVALUE$$

The following operators are used for the corresponding operators in matlab expressions:

z

$$\boldsymbol{and}_m,\ \boldsymbol{or}_m,\ .\widehat{\ }_m\ : PVALUE$$

The following are unary operators occurring in matlab expressions:

z

$\boldsymbol{mp_m}, \boldsymbol{mm_m}, \boldsymbol{\sim_m} : PVALUE$

The following relations and operations over reals are provided to give a distinct priority to occurrences of these operations in Fcn expressions:

z

$+_{\boldsymbol{f}}, -_{\boldsymbol{f}}, *_{\boldsymbol{f}}, /_{\boldsymbol{f}} \ : PVALUE$

The following relations over reals (returning reals) are used as target for the corresponding operators in fcn expressions:

z

$=_{\boldsymbol{f}}, \neq_{\boldsymbol{f}}, <_{\boldsymbol{f}}, >_{\boldsymbol{f}}, >=_{\boldsymbol{f}}, <=_{\boldsymbol{f}} : PVALUE$

z

$\boldsymbol{and_f}, \boldsymbol{or_f}, \hat{}_{\boldsymbol{f}} : PVALUE$

The following are unary operators occurring in Fcn expressions:

z

$\boldsymbol{mp_f}, \boldsymbol{mm_f}, \boldsymbol{not_f} : PVALUE$

3.6 Z Specifications

The following greatly simplified abstract syntax for Z suffices for describing the output from the translator.

Given sets are introduced to designate the words and decorations from which Z identifiers are formed.

z

$[\boldsymbol{WORD}, \boldsymbol{DECOR}]$

An identifier is then defined:

z

IDENT

$\boldsymbol{word}: WORD; \boldsymbol{decor}: DECOR$

One specific decoration is declared, the empty decoration:

z

$\boldsymbol{DecorEmpty}: DECOR$

The following names are introduced for specific distinct identifiers which are declared either in the ProductZ libraries or in the ClawZ Z Library and are used in the specifications generated by ClawZ.

Ui is the Z name for the identifier "$\mathbb{U}$" which is the name of a constant declared in the ProofPower-Z library. This constant is useful in declaring names in Z specification whose type is not known but can be inferred from the context, and it is used by the ClawZ translator in input and output port declarations.

Ei is the identifier of the ProofPower-Z constant "e" used in forming real values.

S2Ui is the identifier "S2U" of an injection from Z real values into some unified value space.

V2Ui is the identifier "V2U" of an injection from sequences of Z real values into some unified value space.

M2Ui is the identifier "M2U" of an injection from sequences of sequences of Z real values into some unified value space.

INeg is the identifier " $\sim$ " of unary negation over the integers in ProofPower-Z.

RNeg is the identifier " $\sim_R$" of unary negation over the real numbers in ProofPower-Z.

Composei is the identifier "o" of relational composition.

Slicei is the identifier " $:_m$".

R2zi is the identifier "*r2z*".

The injections into the unified value space are used in transmission of unified parameter values from Simulink models into the resulting Z specification (see: 3.7.2) when the "Unified" parameter type is specified. Corresponding Z declarations would be required in the ClawZ Z Library ([6], [7]) or in a supplement if the Unified parameter type were to be fully supported, but these have not yet been commissioned.

```
z
│    Ui, Ei, S2Ui, V2Ui, M2Ui, INeg, RNeg,
│    Slicei, Composei, R2zi: IDENT
```

The following names are introduced for specific distinct *WORD*s which are declared either in the ProductZ libraries or in the ClawZ Z Library and are used in the specifications generated by ClawZ.

Active is the Z name for the word "Active" which is the name of a schema declared in the ProofPower-Z library which tests whether the *Action?* port is set.

Inactive is the Z name for the word "Inactive" which is the name of a schema declared in the ProofPower-Z library which tests whether the *Action?* port is clear.

```
z
│    Active, Inactive: WORD
```

The following definition of *Z_EXPR* is highly simplified and application oriented:

z

$$
\begin{array}{lll}
\boldsymbol{Z\_EXPR} ::= & & \boldsymbol{SchemaRef}\ \ WORD \\
& | & \boldsymbol{Ident}\ IDENT \\
& | & \boldsymbol{Application}\ (Z\_EXPR \times Z\_EXPR) \\
& | & \boldsymbol{Selection}\ (Z\_EXPR \times Z\_EXPR) \\
& | & \boldsymbol{BindingDisplay}\ (\mathbb{F}\ (IDENT \times Z\_EXPR)) \\
& | & \boldsymbol{PvalueZexpr}\ PVALUE \\
& | & \boldsymbol{ZSequence}\ (seq\ Z\_EXPR) \\
& | & \boldsymbol{ZPair}\ (Z\_EXPR \times Z\_EXPR) \\
& | & \boldsymbol{ZLambdaExp}\ [\boldsymbol{decl}: seq\ Z\_DEC; \boldsymbol{exp}: Z\_EXPR] \\
& | & \boldsymbol{ZLambdauU}\ Z\_EXPR \\
& | & \boldsymbol{ZInfixOps}\ (Z\_EXPR \times seq(IDENT \times Z\_EXPR)) \\
& | & \boldsymbol{ZBrackets}\ Z\_EXPR \\
& | & \boldsymbol{ZBus}\ (seq\ Z\_EXPR) \\
& | & \boldsymbol{ZHSchema}\ [\boldsymbol{decl}: seq\ Z\_DEC; \boldsymbol{pred}: Z\_PRED] \\
& | & \boldsymbol{ZTheta}\ Z\_EXPR \\
& | & \boldsymbol{ZNat}\ \mathbb{N} \\
& | & \boldsymbol{ZSConj}\ (Z\_EXPR \times Z\_EXPR) \\
& | & \boldsymbol{ZSDisj}\ (Z\_EXPR \times Z\_EXPR) \\
\& & & \\
\boldsymbol{Z\_DEC} ::= & & \boldsymbol{DecSchemaRef}\ \ WORD \\
& | & \boldsymbol{DecSchema}\ \ Z\_EXPR \\
& | & \boldsymbol{DecDec}\ [\boldsymbol{names}: seq\ IDENT; \boldsymbol{type}: Z\_EXPR] \\
& & \\
\& & & \\
\boldsymbol{Z\_PRED} ::= & & \boldsymbol{PredEq}\ (Z\_EXPR \times \mathbb{F}_1\ Z\_EXPR) \\
& | & \boldsymbol{PredBool}\ Z\_EXPR \\
& | & \boldsymbol{PredConj}\ (\mathbb{F}\ Z\_PRED) \\
& | & \boldsymbol{PredDisj}\ (\mathbb{F}\ Z\_PRED)
\end{array}
$$

The “PvalueZexpr” clause is used where a Simulink block parameter value which is legal Z is to be used as part of the output Z specification.

The “ZBus” clause is semantically equivalent to distributed concatenation, and should be translated on output as the distributed concatenation of the sequence display formed from the arguments. It is used only where the sequence of values corresponds to a sequence of signals which forms a bus (and has not been subject to any simplifications which obscure the correspondence with the bus structure). It plays a role in the simplifications which are undertaken during virtualization (see section 6.7).

The arguments to the *PredEq* constructor are respectively:

1 a single ”source” value (usually the source of a line in a simulink model)

2 a set of destination values (usually the destinations of a line in a simulink model)

When pretty printed the source item should be shown as the rightmost operand of the equation.

A basic declaration is either a schema reference, a schema exression, or an identifier-list/type pair.

A declaration is a sequence of basic declarations.

z
$$\boldsymbol{Z\_DECL} \mathrel{\widehat=} seq\ Z\_DEC$$

A predicate is a set of equations, where each equation is a set of at least two *Z_EXPR*, one of which is distinguished (the output port driving a line, which is distinguished so that it can be placed at one end of the equation).

Three kinds of Z paragraph are required, a schema definition, a schema box and an abbreviation definition. Abbreviation definitions are used in the translation of Matlab .m files.

z
$$\boldsymbol{Z\_SCHEMADEF} \mathrel{\widehat=} [\boldsymbol{name}\colon WORD;\ \boldsymbol{value}\colon Z\_EXPR]$$

z
$$\boldsymbol{Z\_HSCHEMA} \mathrel{\widehat=} [\boldsymbol{decl}\colon Z\_DECL;\ \boldsymbol{pred}\colon Z\_PRED]$$

z
$$\boldsymbol{Z\_SCHEMABOX} \mathrel{\widehat=} [\boldsymbol{name}\colon WORD;\ \boldsymbol{decl}\colon Z\_DECL;\ \boldsymbol{pred}\colon Z\_PRED]$$

z
$$\boldsymbol{Z\_ABBREVDEF} \mathrel{\widehat=} [\boldsymbol{ident}\colon IDENT;\ \boldsymbol{value}\colon Z\_EXPR]$$

z
$$\begin{array}{lll} \boldsymbol{Z\_PARA} & ::= & \boldsymbol{SchemaDef}\ Z\_SCHEMADEF \\ & | & \boldsymbol{SchemaBox}\ Z\_SCHEMABOX \\ & | & \boldsymbol{AbbrevDef}\ Z\_ABBREVDEF \end{array}$$

A Z specification is a finite sequence of Z paragraphs.

z
$$\boldsymbol{Z\_SPEC} \mathrel{\widehat=} seq\ Z\_PARA$$

Since there is some discretion in the ordering of the paragraphs and at a later date it might be required to give the user some control over the ordering, a representation of the paragraphs in the Z specification resulting from translating a Simulink model is provided which encapsulates the minimum constraint on their ordering in the final specification.

The blocks which are the immediate constituents of a system or subsystem must be defined before the system itself, in any order. These are held as a mapping indexed by the names of the blocks.

The following free-type provides a representation which contains only this ordering information.

z

$\boldsymbol{SYS\_SPEC} ::= \boldsymbol{SysSpec}\ [\boldsymbol{block\_specs}: PVALUE \mathrel{+\!\!\!\twoheadrightarrow} SYS\_SPEC; \boldsymbol{z\_spec}: Z\_SPEC]$

Henceforth the block specs will be referred as having the following type (which could not be used initially because of the required recursion).

z

$\boldsymbol{BLOCK\_SPECS} \mathrel{\widehat{=}} PVALUE \mathrel{+\!\!\!\twoheadrightarrow} SYS\_SPEC$

The block specifications are shown as a map and can be output in any order, but must precede the specification of the system built from those blocks.

3.7 Library Metadata

3.7.1 Metadata Content and Representation

The library metadata files provide information necessary to the translator in relating the Simulink library blocks to the Z libraries which are their formal specifications.

In this version of ClawZ only two kinds of Z specification will be supported:

1. A schema definition in which the input and output ports follow a conventional naming scheme.
2. A function which takes a binding of parameters as its argument and returns a schema of the above form.

It is desirable to be able to model a Simulink library block by a number of distinct schemas, since the parameterisation of library blocks provides an effective overloading of operators which is awkward to realise in a strictly typed language without overloading.

The library metadata is therefore permitted to specify not only the block type, but also any other block parameter as indicating the use of a particular schema. In this way, for example, the number of ports can be used in selecting the applicable Z definition. In extremis a user could provide a specific translation of a single instance of a library block selected by the blockname used in his diagram for that instance. Selection by logical position in diagram is given special support by allowing a "path" to be specified, which will be matched against the heirarchic block name while selecting the best library block.

The library metadata also includes information about which block parameters are to be transmitted to parameterised specifications. This will be an empty set of parameters for non-parameterised specifications.

With the introduction of support for action subsystems library blocks having state must also be provided with specifications of how the state is to be reset or held. These are similar in structure to the principle definition of the library block (i.e. their parameterisation is the same and their signatures are a subset of the principal definition) but they have different names which are at the discretion of the library designer. The names of these additional schemas must be supplied in the metadata in a manner similar to that in which the name of the principle definition is given.

As a result of support for translation of artificial subsystems of libraries there may be more than one alternative specification for a single subsystem block in a library. To disambiguate reference to Z specifications for subsystems in artificial subsystems of libraries a parameter supplying the name of the relevant artificial subsystem is included. This will not be present unless the metadata element relates to an artificial subsystem.

The library metadata therefore contains up to ten elements for each Z specification:

1. The name of the artificial subsystem of the library to which this item of metadata relates, known as "ASname" in (the concrete representation of) the metadata.
2. The name of the Z specification, known as "Zname". This will either be the name of a schema or of a function yielding a schema when supplied with a suitable binding of parameters.
3. The name of the Z specification for holding the state, known as "HeldZname".
4. The name of the Z specification for resetting the state, known as "ResetZname".
5. An optional "BlockPath" parameter, which must match against the heirarchic name of the block being translated.
6. A set of parameters known as "SelectionParameters" which must be present with specific values in the parameters for an instance of a Simulink library block for the specification to be used in translating that instance.
7. A set of parameters known as "TransmittedParameters" which indicates the block parameters which are to be passed in a binding as an argument to the Z specification. A translation code is provided with each parameter indicating how the parameter supplied to the Simulink library block should be processed before inclusion in the Z specification. This may be omitted from the metadata, in which case the Z specification is not parameterised. In the abstract representation this case is indicated by the use of an empty set of transmitted parameters.
8. An optional parameter known as "InputPortTypes", which indicates the types of the input ports on the block. This is a string with one entry for each input port in ascending order with no gaps. Each entry must be "S", "V" optionally followed by a numeral, "G", or "U" standing for port types *ScalarPT*, *VectorPT*, *GenericPT*, *UnknownPT* respectively. If the string is shorter than the number of input ports then the unspecified ports are taken as unknown. The numeral suppled for a vector signal indicates the width of the signal, its absence or the value "0" indicating that the width is unknown. 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, that they all have the same generic type (and therefore the same type when instantiated), and will if this is a vector type also have the same length.
9. An optional parameter known as "OutputPortTypes", which indicates the types of the output ports on the block, using the same format as the "InputPortTypes" parameter.

10. An optional parameter known as "UsedLocalVariables" which provide the names of all the local variables which are used in this block and will therefore need to be passed to it as parameters. Local variables are those which are mask variables for some enclosing masked subsystem. When a subsystem in a library is invoked by block reference these local variables are effectively rebound to their values at the point of reference.

The grammar for "InputPortTypes" and "OutputPortTypes" and "UsedLocalVariables" is shown in section 2.7.

A pattern for a block path is specified as follows:

z

$$\boldsymbol{PAT\_EL} ::= \boldsymbol{StarPat} \mid \boldsymbol{QueryPat} \mid \boldsymbol{PlainPat}\ PVALUE$$

Where *StarPat* matches any sequence of names, *QueryPat* matches any single name and *PlainPat* matches a single specific name.

z

$$\boldsymbol{PATTERN} \mathrel{\widehat{=}} seq\ PAT\_EL$$

z

$$\begin{array}{lll}
\boldsymbol{PORT\_TYPE} ::= & & \\
 & & \boldsymbol{ScalarPT} \\
 & \mid & \boldsymbol{VectorPT}\ \mathbb{N} \\
 & \mid & \boldsymbol{GenericPT} \\
 & \mid & \boldsymbol{UnknownPT} \\
 & \mid & \boldsymbol{BusPT}\ (\mathbb{N} \times seq\ [line\_name : PVALUE; port\_type : PORT\_TYPE])
\end{array}$$

A library metadata element is therefore represented in our abstract syntax as follows:

z

$$\begin{array}{l}
\boldsymbol{META\_ELEMENT} \\
\hline
\quad \boldsymbol{as\_name} : PVALUE; \\
\quad \boldsymbol{z\_name} : IDENT; \\
\quad \boldsymbol{held\_z\_name} : OPT[IDENT]; \\
\quad \boldsymbol{reset\_z\_name} : OPT[IDENT]; \\
\quad \boldsymbol{block\_path} : PATTERN; \\
\quad \boldsymbol{select\_pars} : \mathbb{F}_1\ PARAM; \\
\quad \boldsymbol{transmit\_pars} : \mathbb{F}\ PARAM; \\
\quad \boldsymbol{input\_port\_types} : seq\ PORT\_TYPE; \\
\quad \boldsymbol{output\_port\_types} : seq\ PORT\_TYPE; \\
\quad \boldsymbol{used\_maskvars} : \mathbb{F}\ PVALUE \\
\hline
\end{array}$$

The *as_name* component will be set to *NullString* when reading metadata if the *ASname* item is absent. On output the *ASname* field will be omitted if the *as_name* component has value *NullString*.

A library metadata file is simply a sequence of metadata elements.

z
$$\boldsymbol{META\_FILE} \mathrel{\widehat{=}} seq_1\ META\_ELEMENT$$

3.7.2 Parameter Format and Representation

Translation of a parameter yields a result of the form:

z
$$\begin{array}{lll}\boldsymbol{TRANSLATION\_RESULT} ::= & \boldsymbol{TMatch}\ Z\_EXPR \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad | & \boldsymbol{TNoMatch} \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad | & \boldsymbol{TFail}\end{array}$$

The *TMatch* values are used to return the value resulting from a sucessful parameter translation. When a translation fails it can return either *TNoMatch*, causing the match to fail, and indicating that the parameter supplied was not of the required kind, or it can return a *TFail* where a translation method, perhaps for some kind of expression, is unable to complete the translation but where there is no reason to doubt that the match was good and search for another match.

A set of names for specific *PVALUE*s to be used as translation codes are now introduced:

Quoted is the parameter value “Quoted”. If this translation code is used the parameter is to passed to the Z specification as a quoted string.

Unquoted is the parameter value “Unquoted”. If this translation code is used the parameter is to passed to the Z specification with any outer quotation marks removed but with no other changes. Support for the historical parameter code ”Numeric” is no longer required.

Scalar is the parameter value “Scalar”. If this translation code is used the parameter is assumed to be a matlab scalar expression limited to the grammar shown in section 2.4.4.

Vector is the parameter value “Vector”. The parameter is assumed to be a Matlab expression denoting a numeric row vector or scalar and is to be translated into a Z sequence display.

Matrix is the parameter value “Matrix”. The parameter is assumed to be a Matlab expression denoting a numeric matrix, vector or scalar and is to be translated into nested Z sequence displays.

SVM is the parameter value “SVM”. This is not *required* in the model translator but is specified for use in the .m file translator. For this parameter code a parameter is interpreted as either a scalar, a vector or a matrix in that order of priority. The Z type of the resulting expression will depend on which interpretation is taken.

Unified is the parameter value “Unified”. This is the same as SVM except that the resulting parameter is coerced into a single value space by the use of the functions “S2U”, “V2U”, and “M2U” respectively.

Fcn is the parameter value "Fcn". This is similar to "Scalar" except that special treatment is given to the array named "u", and the resulting expression is abstracted over "u" before being supplied to the selected library block.

Checkbox is the parameter value "Checkbox". This parameter type corresponds to the checkbox masked variable type. A parameter of type *Checkbox* must be either "off" or "on", which are translated into the numbers zero and one respectively.

z

$$\boldsymbol{Quoted}, \boldsymbol{Unquoted}, \boldsymbol{Scalar}, \boldsymbol{Vector},$$
$$\boldsymbol{Matrix}, \boldsymbol{SVM}, \boldsymbol{Unified}, \boldsymbol{Fcn}, \boldsymbol{Checkbox}: PVALUE$$

The translation codes *Quoted* and *Unquoted* never fail, unconditionally transmitting the parameter into the Z specification, with or without outer quotes as required, but with no other alterations.

The translation codes *Scalar*, *Vector*, *Matrix*, *SVM*, *Unified*, and *Fcn* invoke a futher parse of the parameter using one of the grammars specified in section 2.4. If the parse fails or if the resulting object is of higher dimension than that requested, the match fails. Otherwise the parameter is subject to a simple transformation and output as a Z object

z

$$\begin{array}{lll} \boldsymbol{PARAMETER} ::= & & \\ & & \boldsymbol{ParName}\ PVALUE \\ & | & \boldsymbol{ParScalar}\ SEXP \\ & | & \boldsymbol{ParVector}\ VEXP \\ & | & \boldsymbol{ParMatrix}\ MEXP \end{array}$$

The following types represent the results of parsing the various mask variable parameters.

z

$$\begin{array}{lll} \boldsymbol{MASK\_STYLE} ::= & & \\ & & \boldsymbol{MSEdit} \\ & | & \boldsymbol{MSCheckbox} \\ & | & \boldsymbol{MSPopup}\ (seq\ PVALUE) \end{array}$$

The result of parsing a *MaskVariables* parameter should be a sequence which maps variable numbers to variable names.

z

$$\boldsymbol{MASK\_VAR\_PAR} \mathrel{\widehat=} seq\ PVALUE$$

z

$$\boldsymbol{MASK\_STYLE\_PAR} \mathrel{\widehat=} seq\ MASK\_STYLE$$

z

$$\boldsymbol{MASK\_VALUE\_PAR} \mathrel{\widehat=} seq\ PVALUE$$

Special translation of a parameter yields a result of the form:

z

$$\boldsymbol{SPECIAL\_RESULT} ::= \boldsymbol{SRScalar}\ Z\_EXPR \mid \boldsymbol{SRVector}\ Z\_EXPR \mid \boldsymbol{SRFail}$$

3.8 Steering File Structure

The ClawZ steering file is in the same structured format as that used for models and is therefore read in as a *STRUCTURE*.

Only the following four different *PNAMES* may be used at the top level of this structure:

InPortTypes is the parameter name "InPortTypes", which is used as the name of a (non-standard) structure in a ClawZ steering file to record the types of the input ports for some block in the model specified by path.

NameMapping is the parameter name "NameMapping", which is used as the name of a structure in a ClawZ steering file used to control the Simulink to Z name mapping.

OutPortTypes is the parameter name "OutPortTypes", which is used as the name of a (non-standard) structure in a ClawZ steering file to record the types of the output ports for some block in the model specified by path.

VariableTypes is the parameter name "VariableTypes", which is used as the name of a structure in a ClawZ steering file to record the types of matlab variables used in the model.

z

$$\boldsymbol{InPortTypes}, \boldsymbol{NameMapping}, \boldsymbol{OutPortTypes}, \boldsymbol{VariableTypes} : PNAME$$

3.9 Matlab Variable Types

During the translation of matlab .m files "types" are assigned to the variables based on the structure of the expression assigned to the variable. These types are later used during the type inference stage of model translation.

A type will either be scalar, vector or matrix, but the length is required where this can be derived. Allowing for some future generalisation to higher dimensions of arrays an appropriate representation of the required information would simply be a seqence of numbers, whose length will in fact be 0, 1 or 2 and whose values are either the size of the relevant dimension of the array or zero if it cannot be deduced. The specification provides at present only for limited intelligence in determining the sizes.

z

$\boldsymbol{MVARTYPE} \mathrel{\widehat{=}} seq\ \mathbb{N}$

The values are accumulated in a mapping which assigns *MVARTYPE*s to *PVALUE*s. The name used in this mapping is the name after application of any prefix or suffix.

z

$\boldsymbol{MVARTYPES} \mathrel{\widehat{=}} PVALUE \mathrel{+\!\!\!\twoheadrightarrow} MVARTYPE$

3.10 ClawZ Control Data

When the ClawZ program is invoked various information is supplied as parameters.

This includes:

1. the pathname of the model or library file to be compiled
2. the pathname of a metadata file
3. the pathname of a steering file
4. an optional filename for metadata output (if this filename is supplied the translation will be undertaken as a library translation not a model translation)
5. a list of output file paths and details of which parts of the translated model are to be written to each file
6. a list of artificial subsystem specifications, and for each such artificial subsystem details of how the output is to be written into files.

The following Z data structures model this information.

OUTPUT_FILTER_SPEC is used to represent a prescription for a subset of the output of the translator to be written to a particular file.

The information to be written to the file whose pathname is *output_file*, is all the Z output from blocks or subsystems in the model whose pathnames match a pattern in *excl* but do not match one in *incl* and have not previously been output (to some other file).

z

OUTPUT_FILTER_SPEC

$\boldsymbol{output\_file}: STRING;$

$\boldsymbol{excl}, \boldsymbol{incl}: seq\ PATTERN$

Artificial subsystem specifications permit the synthesis of subsystems involving parts of the subsystems in the model.

To specify artificial subsystems we need a structure *BLOCK_MODIFIER_SPEC* which specifies a modification to a part of a subsystem. The component *path* is the path to a block in the model, i.e. a sequence of subsystem names. The component *mod* specifies the required change, using the free type *BLOCK_MODIFIER*.

There are three kinds of modification which are permitted. The constructor *Include* provides a list of blocknames of blocks which are to be omitted from this subsystem (all others will be included). The constructor *Exclude* provides a list of blocknames of blocks which are to be omitted from this subsystem (all others will be included). The constructor *ASname* provides the name of an artificial subsystem of a library which is required to be used in satisfying a block reference to a subsystem in that library. *Include* and *Exclude* modifications are only permitted for paths which are the paths of subsystems, *ASname* modifications are permitted only for paths which are the paths of block references.

z

$$
\begin{array}{lll}
\boldsymbol{BLOCK\_MODIFIER} ::= & & \boldsymbol{Include}\ (\mathbb{F}\ PVALUE) \\
 & | & \boldsymbol{Exclude}\ (\mathbb{F}\ PVALUE) \\
 & | & \boldsymbol{ASname}\ PVALUE
\end{array}
$$

z

$\boldsymbol{FullFilter} \mathrel{\widehat=} Exclude\ \{\}$

z

$\boldsymbol{EmptyFilter} \mathrel{\widehat=} Include\ \{\}$

Both for block selection and for output selection these paths may be matched against patterns supplied by the user, however no pattern matching is involved in their use here.

z

```
┌─BLOCK_MODIFIER_SPEC──────────────────────────
│    path: PATTERN;
│    filter: BLOCK_MODIFIER
└──────────────────────────────────────────────
```

Each artificial subsystem has a *name* which is used as the initial part of the names of the Z specifications created for this subsystem. The artificial subsystem is specified as having a top-level which is a subset of an identified system or subsystem of the model, and by specifying optionally a number of subsetting operations on subsystems directly or indirectly contained in that top level subsystem. The path of the *top* *BLOCK_MODIFIER_SPEC* must be absolute, the paths in the *rest* of the *BLOCK_MODIFIER_SPEC*s must be relative to that path.

z

```
┌─ART_SUBSYS_SPEC──────────────────────────────
│    name: PVALUE;
│    top: BLOCK_MODIFIER_SPEC;
│    rest: seq BLOCK_MODIFIER_SPEC;
│    output_spec: seq OUTPUT_FILTER_SPEC
└──────────────────────────────────────────────
```

We are now able to specify the parameters required for a run of ClawZ.

z
$$
\begin{array}{|l}
\hline
\boldsymbol{RUN\_PARAMS} \\
\quad \boldsymbol{model\_file}, \boldsymbol{meta\_file}, \boldsymbol{steer\_file}: STRING; \\
\quad \boldsymbol{meta\_output}: OPT[STRING]; \\
\quad \boldsymbol{output\_spec}: seq\ OUTPUT\_FILTER\_SPEC; \\
\quad \boldsymbol{art\_subsys\_specs}: seq\ ART\_SUBSYS\_SPEC \\
\hline
\end{array}
$$

We also specify the parameters required for a run of the ClawZ .m file translator.

z
$$
\begin{array}{|l}
\hline
\boldsymbol{M\_FILE\_RUN\_PARAMS} \\
\quad \boldsymbol{m\_file}: STRING; \\
\quad \boldsymbol{output\_file}, \boldsymbol{types\_file}: OPT[STRING]; \\
\quad \boldsymbol{parameter\_type}: PVALUE \\
\hline
\end{array}
$$

3.11 Context Structures

In this section are defined data types which are used in the context for any of the passes over the intermediate model.

The following data type provides the contextual information required for deciding whether to generate state held and/or reset schemas for subsystems and also for deciding which of the available schemas for library blocks should be instantiated.

held the context is *HCHeld* if the block is an action subsystem of which the relevant port is set to "held" the state when disabled, or if it is enclosed in an action subsystem and the smallest enclosing action subsystem is set to hold the state.

reset the context is *HCReset* if the block is an action subsystem of which the relevant port is set to "reset" the state when inactive, or if it is enclosed in an action subsystem and the smallest enclosing action subsystem is set to reset the state.

void the context is *HCVoid* if the block is not in a library compilation and neither is nor is contained in an action subsystem.

unknown the context is *HCUnknown* if the block is in a library compilation and is not an action subsystem.

The *HOLD_CONTEXT* influences the generation of state held and state reset schemas. Contexts *HCHeld* and *HCReset* indicate that held and reset schemas are required (respectively) if the subsystem has any state. If the context is *HCUnknown* both held and reset schemas are required to be generated, if *HCVoid* neither.

z

```
HOLD_CONTEXT ::=
            HCHeld
        |   HCReset
        |   HCUnknown
        |   HCVoid
```

3.12 Intermediate Model Representation

We here define the data structures representing information derived from the Simulink model which is necessary in defining the transformation into Z. The analysis which gives values to these structures is defined in section 6.6.

z

```
┌─PORT_DETAILS──────────────────────────
│    line_name: PVALUE;
│    port_type: PORT_TYPE
└───────────────────────────────────────
```

z

```
┌─PORT_INFO─────────────────────────────
│    input_port_details: PVALUE ⇸ PORT_DETAILS;
│    output_port_details: PVALUE ⇸ PORT_DETAILS
└───────────────────────────────────────
```

The following key type is used in sorting the declarations in a schema into the required order. The number on *ActionInv* is used for action ports on Merge blocks and is set to zero for the action port on an action subsystem.

z

```
INVKEY ::=
            NoInv
        |   ActionInv ℕ
        |   EnableInv
        |   TriggerInv
        |   InportInv ℕ
        |   OtherInv IDENT
        |   OutportInv ℕ
```

In the following the three declaration lists are for the main definition and for state held and state reset (which will often be empty).

z

$\boldsymbol{INVOCATION} \mathrel{\widehat=} INVKEY \times (Z\_DECL \times Z\_DECL \times Z\_DECL)$

The following projection functions are defined for use on invocations.

z

$\boldsymbol{main\_inv}, \boldsymbol{held\_inv}, \boldsymbol{reset\_inv}$
$: (Z\_DECL \times Z\_DECL \times Z\_DECL) \rightarrow Z\_DECL$

$\forall\ m, h, r: Z\_DECL \bullet$
$main\_inv\ (m, h, r) = m$
$\land\quad held\_inv\ (m, h, r) = h$
$\land\quad reset\_inv\ (m, h, r) = r$

z

$\boldsymbol{VIRTUAL} ::=$
$\quad \boldsymbol{VUnknown}$
$\mid\quad \boldsymbol{VInhibit}$
$\mid\quad \boldsymbol{Virtual}\ (PVALUE \twoheadrightarrow Z\_EXPR)$

The following schema contains those components which are common to subsystems and library blocks.

z

COMMON_INFO

$\boldsymbol{specification}: Z\_SPEC;$
$\boldsymbol{invocation}: INVOCATION;$
$\boldsymbol{virtual}: VIRTUAL;$
$\boldsymbol{used\_maskvars}: \mathbb{F}\ PVALUE$

The following schema contains the information required for library blocks.

z

BLOCK_INFO

$\boldsymbol{pars}: \mathbb{F}\ PARAM;$
$\boldsymbol{input\_port\_types}: PVALUE \twoheadrightarrow PORT\_TYPE;$
$\boldsymbol{output\_port\_types}: PVALUE \twoheadrightarrow PORT\_TYPE;$
$COMMON\_INFO$

The following structure documents an "action complex", which is an *If* or a *SwitchCase* block, the action subsystems to which it is connected and the *Merge* blocks to which they are in turn connected. It serves two primary purposes. The first is to permit the checking of the specified constraints on these systems and hence to enable appropriate error reporting. The second is to provide the necessary information for the supplementary wiring which is required to feed the values on subsystem action ports to the appropriate *Merge* block.

The terminology used in the component names is as follows:

root	This is the blockname of the *If* or a *SwitchCase* block.
type	This is the blocktype of the root block.
tail	This is the blockname of the *Merge* block.
open	This is *true* if the block is an *If* block without an *else* clause or if it is a *SwitchCase* block without a *default*. If it is *true* this is reported as an error in the first phase, prior to *Merge* synthesis. When *Merge* synthesis is implemented open action complexes will be permitted and a *Merge* block with state will be implemented.
subsys_map	This maps portnames on the root block to the blockname and portname of the action subsystem to which it is connected.
merge_map	This maps a blockname and output portname of each action subsystem to the blockname and portname on the Merge block to which it is connected.

z
```
┌─ACTION_COMPLEX──────────────────────────
│   root, type, tail: PVALUE;
│   open: BOOL;
│   subsys_map, merge_map: PVALUE ⇸ PVALUE
└─────────────────────────────────────────
```

The *action_info* information required for subsystems consists of a flag which is set if this is an action subsystem, the set of action complexes which occur in the subsystem, and the held context.

z
```
┌─ACTION_INFO─────────────────────────────
│   action_subsys: BOOL;
│   complexes: 𝔽 ACTION_COMPLEX;
│   held_context: HOLD_CONTEXT
└─────────────────────────────────────────
```

z
```
│   initial_action_info: HOLD_CONTEXT → ACTION_INFO
├──────────────────────
│   ∀hc: HOLD_CONTEXT•
│
│   initial_action_info hc =
│         (action_subsys ≙ false,
│         complexes ≙ {},
│         held_context ≙ hc)
```

z
```
┌─ SUBSYS_INFO ─────────────────────────────
│  subpars, syspars: 𝔽 PARAM;
│  lines: 𝔽 LINE;
│  mv_ctxt: 𝔽 PVALUE;
│  action_info: ACTION_INFO;
│  COMMON_INFO
└───────────────────────────────────────────
```

z
```
┌─ A_LIB_BLOCK ─────────────────────────────
│  block_info: BLOCK_INFO;
│  port_info: PORT_INFO
└───────────────────────────────────────────
```

z
```
│A_BLOCK  ::=   ALibBlock A_LIB_BLOCK
│          |    ASubsys [  subsys_info: SUBSYS_INFO;
│                          port_info: PORT_INFO;
│                          blocks: PVALUE ⇸ A_BLOCK    ]
```

z
```
┌─ A_SUBSYS ────────────────────────────────
│  subsys_info: SUBSYS_INFO;
│  port_info: PORT_INFO;
│  blocks: PVALUE ⇸ A_BLOCK
└───────────────────────────────────────────
```

3.13 Global Data

The specification is mostly functional, but to contain the complexity of the information structures passed as parameters some global data has been introduced.

3.13.1 The Steering File

The content of the steering file is accessed from the following global variable, which should contain results of parsing the ClawZ steering file.

z
```
│  steering_file: STRUCTURE
```

3.13.2 Name Translation Table

The following mapping may be set by the user (using function *set_translation_table*), and will otherwise default to something sensible, and may be used in the translation of Simulink block names into Z.

z

$$\boldsymbol{z\_translation\_table}: CHAR \pfun seq\ CHAR$$

The character set used in the destination should be confined to characters which are legal in Z identifiers according to the ProofPower-Z lexis, but this will be assumed not checked.

A new function called *set_translation_table* will be supplied, taking a list of pairs of strings. The first of each pair must be a single character and the second will be a sequence of characteres to which that character must be translated.

z

$$\boldsymbol{set\_translation\_table}: (CHAR \pfun seq\ CHAR) \rightarrow \{0\}$$

3.13.3 String Controls

The following two string controls allow the *ClawZ* user to apply a prefix and/or a suffix to a matlab variable name when it is translated into Z in a .m file or in a Simulink model.

z

$$\boldsymbol{z\_name\_prefix}, \boldsymbol{z\_name\_suffix}: seq\ CHAR$$

They should both default to empty strings.

The following controls influence the translation of Simulink paths into Z words.

z

$$\boldsymbol{z\_name\_filler}: seq\ CHAR;$$
$$\boldsymbol{z\_path\_separator}: CHAR$$

If *z_name_filler* is set to a non empty string then its value will replace all non alphanumeric characters when Simulink blocknames are translated, except for spaces which will be discarded. If *z_name_filler* is set to the empty string then Simulink blocknames will be translated using the translation table *z_translation_table*. For a fuller specification of the translation algorithms see section 4.1.3.

The *z_path_separator* control must be a single character and is used for separating blocknames in the Z translation of a Simulink path.

The following string controls determine the suffixes used for the various additional schema names which have been introduced for translating action subsystems.

The active suffix is used for the schema defining the behaviour of an action subsystem when the action port is set. It defaults to "$_a$".

The held suffix is used for the schema defining the behaviour of a block when it is inactive and holding its state. It defaults to "$_h$".

The reset suffix is used for the schema defining the behaviour of a block when it is inactive and resetting its state. It defaults to "$_r$".

Though the held and reset conditions are intended to determine the state when an action subsytem is activated after being inactive, this effect is achieved by setting the after state appropriately throughout the period when the block is inactive.

z
$$\begin{array}{|ll}
\boldsymbol{active\_suffix}: & seq\ CHAR;\\
\boldsymbol{held\_suffix}: & seq\ CHAR;\\
\boldsymbol{reset\_suffix}: & seq\ CHAR
\end{array}$$

It is intended that these suffixes are used to add a subscript, and that subscripts do not appear elsewhere in Z identifiers. If these conditions do not obtain there is a risk that the algorithms used to ensure uniqueness of Z identifiers will not be effective and that clashes may arise.

3.13.4 Flags

The *double_separator* flag determines whether one or two instances of the *z_path_separator* are used as a separator between blocknames in the translation into Z of a Simulink block path.

z
$$\begin{array}{|l}
\boldsymbol{double\_separator}: BOOL
\end{array}$$

The *virtualize* control determines whether ClawZ attempts to virtualize blocks. It defaults to `false`.

z
$$\begin{array}{|l}
\boldsymbol{virtualize}: BOOL
\end{array}$$

The *inhibit_output_on_error* control determines whether ClawZ inhibits the output of Z specifications subsequent to discovering and reporting an error. It defaults to `true`.

z
$$\begin{array}{|l}
\boldsymbol{inhibit\_output\_on\_error}: BOOL
\end{array}$$

4 NAME, PARAMETER AND .m FILE TRANSLATION

4.1 Name Translation

4.1.1 Primitive Model to Z Translations

We require a convention for converting port numbers to Z IDENTs. Informally the proposed convention is that input port x has name "Inx?" and output port x has name "Out$x$!".

We also mention a function *strip_pval* which takes a quoted PVALUE and strips off the outer quotes (leaving other *PVALUE*s untouched) and *quote_pval* which takes an unquoted PVALUE and adds outer quotes.

z

$$\boldsymbol{strip\_pval}: PVALUE \to PVALUE;$$
$$\boldsymbol{quote\_pval}: PVALUE \to PVALUE$$

4.1.2 Name Conversions

The names written into Z specifications by the translator as Z identifiers (*IDENT*) or words (*WORD*s) come from a variety of different sources and play differing roles in the resulting specifications.

z

$$\boldsymbol{pname2ident}: PNAME \to IDENT;$$
$$\boldsymbol{pvalue2ident}: PVALUE \to IDENT;$$
$$\boldsymbol{change\_mlname}: PVALUE \to PVALUE$$

$$\forall pv: PVALUE \bullet change\_mlname\ pv$$
$$= sc2pv(z\_name\_prefix \frown (pv2sc\ pv) \frown z\_name\_suffix)$$

pname2ident is the name of a mapping from Simulink parameter names to Z identifiers. This must be the identity function on parameter names which are valid Z names. It clearly cannot be the identity on Simulink names which are not valid Z names, and hence cannot be an injection, and there is therefore some small risk of name clashes. Since the names are used only as component names in schema types used to pass parameters the risk of a clash is thought very low.

pvalue2ident is the name of a mapping from *PVALUE*s to Z identifiers. It will only be used on *PVALUE*s which are valid Z identifiers and makes no change to the value, only to the type. It is used for inserting into the Z specification operators and function names occuring in Matlab and Fcn expressions (after they have already been converted to the chosen Z identifier).

change_mlname is the name of a mapping from Matlab names to optionally decorated names. This is used for generating a Z identifier for a Matlab variable name defined in a .m file. There is a user selectable option to 'decorate' these names by adding a prefix or suffix chosen by the user, but otherwise the names should not be changed by this mapping. This does not necessarily involve a decoration in the technical sense in which that term is used in Z. This mapping is not used for operators or function names in Matlab or Fcn expressions, or for masked variable names.

4.1.3 Simulink Block Name Translation

z

$$\boldsymbol{PATH} \mathrel{\widehat{=}} seq\ PVALUE$$

Two functions are required for converting Simulink blocknames or paths into names suitable for use in the resulting Z specification. Because Simulink blocknames are much less restricted in their character set than Z WORDs, the relationship between the Simulink blockname and the corresponding Z cannot be entirely straightforward.

Access to this mapping is provided through the following two functions. *path2locw* returns a local name only, for use on the left of the colon in a declaration. *path2globw* returns a global name compounded from the local names for each block in the sequence supplied as parameter, using underbar as separator. *path2loci* and *path2globi* are variants which return *IDENT*s rather than *WORD*s.

word2ident converts a *WORD* to an *IDENT* with an empty decoration.

z

$$\boldsymbol{word2ident}\colon WORD \to IDENT$$

$$\forall\ w\colon WORD \bullet$$
$$word2ident\ w = (word \mathrel{\widehat{=}} w,\ decor \mathrel{\widehat{=}} DecorEmpty)$$

z

$$\boldsymbol{path2locw}\colon PATH \rightarrowtail WORD;$$
$$\boldsymbol{path2globw}\colon PATH \rightarrowtail WORD;$$
$$\boldsymbol{path2loci}\colon PATH \rightarrowtail IDENT;$$
$$\boldsymbol{path2globi}\colon PATH \rightarrowtail IDENT$$

$$path2loci = path2locw \mathbin{⨾} word2ident;$$
$$path2globi = path2globw \mathbin{⨾} word2ident$$

path2globw is used for translating the names used for definitions, either of the instantiations of library blocks, of synthesised block definitions, or of the schema boxes encapsulating system or subsystem diagrams.

path2locw is used for translating the name of a library block or of a subsystem block for use in the signature of a diagram specification or in the line equations.

One function obtains returns a global name which translates the entire path, another returns a local name derived from the last name in the path in a manner which is sensitive to the context provided by the remainder of the path. The Z words which result from translation will be unique in the appropriate context, globally unique for the global names, unique within the names local to a Simulink subsystem otherwise. The Z words will be lexically valid Z words provided that the relevant string controls and the translation table are set to characters which are acceptable in Z identifiers. The global names will be compounded from the local names separated by one or two occurrences of the *z_path_separator* according to whether the *double_separator* flag is set, modulo prefixing by "Z" of z words which would otherwise begin with a decimal digit.

The local name translation algorithm is memoised in context, i.e. a table is held for each subsystem in the model in which the names of all the blocks in the subsystem and their translation is held. When a local name is first translated in any particular context it will be translated using the algorithm which follows, but on subsequent occasions the previous translation will be retrieved from the table.

First an "immediate translation" will be obtained as follows:

- The name is translated character by character, using:
  - if *z_name_filler* is a non-empty string, use the identity function for alphanumerics and for the *z_name_separator* character, the empty string for spaces and *z_name_filler* for other characters.
  - the translation table, either as supplied by the user (leaving unchanged values not assigned in the table), or else the default translation table.
- Leading or trailing underbars or *z_name_separator* characters are removed.
- If the *double_separator* flag is set sequences of more than one *z_name_separator* are replaced by one separator, otherwise all occurrences the *z_name_separator* are removed.

Then, if the name begins with a decimal digit, or if it is the empty string, a "Z" will be prepended to the "immediate translation", the result (with or without a Z according to the condition stated) is called the "translation2". Now a check will be made whether translation2 has already been used as the translation of a blockname in this context. If not translation2 will be used as the final translation and that fact recorded in the table. If translation2 has been used in this context (as the translation of some Simulink blockname) then the smallest positive integer numeral which yields an unused Z word when appended to translation2 will be used and recorded as the final translation.

The global name for a path will be obtained as follows. First all the blocknames in the path are translated in the appropriate context as above. The sequence of resulting z words is then modified as follows. For every z word except the first in the sequence which begins with "Z", a check is made whether the immediate translation of the Simulink blockname of which the Z word is the final translation begins with a decimal digit, and if so, remove the initial "Z" from the Z word. The translation of the path is then obtained by appending all the resulting z words separating each by one or two occurrences of the *z_path_separator* according to the setting of the *double_separator* flag.

When artificial subsystems are created the following special considerations apply. Each artificial subsystem has a name and a root path, which is the path to the simulink subsystem which is to be the top level subsystem of the artificial subsystem. The local names used for blocks in the artificial subsystem are to be the same as the local name used for the corresponding block in the main system. The global names are compounded from the local names in the usual way, and will differ from those for the corresponding blocks in the main system because the path in the artificial subsystem differs from the path in the main system (beginning with the artificial subsystem name instead of the initial part of the path which corresponds to the path of the top level of the artificial subsystem).

4.1.4 Suffixing

While translating models or libraries involving action subsystems it is necessary to apply various suffixes to Z identifiers or words to give unique names for the additional definitions required.

The suffixes used are determined by the values of string controls.

A difficulty arises here from a clash between the requirement for action subsystems and the manner in which the prior ClawZ specification is formulated. The ClawZ specification follows Spivey's *The Z Notation* in distinguishing between *WORD*s and *IDENT*s (which are possibly decorated *WORD*s) and in requiring that a schema be given a name which is a *WORD* not an *IDENT*. ProofPower-Z is less fussy than this and allows defined schemas to have decorated names, and this feature is exploited in the requirement for action subsystems.

In order to allow this without too much disruption to the ClawZ specification we disregard the fact that suffixes are strictly decorations and can therefore only be placed on *IDENT*s which are not already decorated and invite the reader to thing of both WORDs and IDENTs as if they were strings and suffixing as appending two additional characters to the string (the first being the %down% character).

The following functions apply the appropriate suffixes and are here specified informally.

z

$$\boldsymbol{w_a}, \boldsymbol{w_h}, \boldsymbol{w_r}\colon WORD \rightarrow WORD;$$
$$\boldsymbol{i_a}, \boldsymbol{i_h}, \boldsymbol{i_r}\colon IDENT \rightarrow IDENT$$

The function names themselves use the relevant default suffixing convention. i.e. the suffix $_a$ variants are to use the suffix in the string control *active_suffix*, the suffix $_h$ variants are to use the suffix in the string control *held_suffix*, and the suffix $_r$ variants are to use the suffix in the string control *reset_ suffix*.

4.2 Parameter Parsing

The following functions are informally defined as performing a parse according to the above specified grammars (section 3.7.2) yielding an appropriate parse tree.

z

$$\boldsymbol{parse\_param}\colon PVALUE \nrightarrow PARAMETER$$

The Fcn expression grammar is parsed by this function:

z

$\boldsymbol{parse\_fcn\_param}: PVALUE \nrightarrow SEXP$

These function should be understood to be semi-formally specified as converting concrete syntax from the grammars in section 2 into the abstract structures defined in section 3.

The *Ports* parameter also has a rudimentary grammar (see section 2.7, a comma separated list of numbers enclosed in square brackets) and needs to be converted to the structure *PORT_PARAM* Values not specified should be taken as zero.

z

$\boldsymbol{parse\_ports\_param}: PVALUE \nrightarrow PORT\_PARAM$

The *OutputSignals* parameter to a *BusSelector* block also has a rudimentary grammar (see section 2.7, a comma separated list of selectors, each of which is a "." separated list of names) and needs to be converted to the structure *OUTPUTSIGNALS_PARAM*

z

$\boldsymbol{parse\_outputsignals\_param}: PVALUE \nrightarrow OUTPUTSIGNALS\_PARAM$

The *Outputs* parameter to a *Demux* block has a rudimentary grammar (see section 2.7, a positive natural number or a comma separated list of positive natural numbers) and needs to be converted to a value of type *OUTPUTS_PARAM*

z

$\boldsymbol{parse\_outputs\_param}: PVALUE \nrightarrow OUTPUTS\_PARAM$

The following functions parse the three parameters associated with mask variables:

z

$\boldsymbol{parse\_maskvar\_param}: PVALUE \nrightarrow MASK\_VAR\_PAR;$
$\boldsymbol{parse\_maskstyle\_param}: PVALUE \nrightarrow MASK\_STYLE\_PAR;$
$\boldsymbol{parse\_maskvalue\_param}: PVALUE \nrightarrow MASK\_VALUE\_PAR$

Note that *parse_maskvar_param* is expected to yield a sequence of mask variable names. The "MaskVariables" parameter consists of a set of name/position pairs, the position is the position in the required sequence.

The following function parses the *popup* parameter translation code. A *popup* parameter translation code it the "Popup" alternative of the grammar for translation codes given in section 2.6.

z

$\boldsymbol{parse\_popup\_tc}: PVALUE \nrightarrow seq\ PVALUE$

The following functions are used in parsing values in metadata files and steering files.

z

$\boldsymbol{parse\_port\_types}: PVALUE \nrightarrow seq\ PORT\_TYPE$

The following function parses a path, which is a pattern without wildcards, when provided as a PNAME.

z

$\boldsymbol{parse\_pathn}: PNAME \nrightarrow PATH$

The following function parses a matlab variable type, which has the syntax $< ports\_param >$, except optionally that the implementation need not tolerate spaces.

z

$\boldsymbol{parse\_var\_type}: PVALUE \nrightarrow seq\ \mathbb{N}$

This differs from parsing a ports parameter proper in the type of the result.

4.3 Expression Translation

4.3.1 Coercions and Literals

The following functions specify the conversion into ProofPower-Z expressions of "Quoted" and "Unquoted" parameters:

z

$\boldsymbol{quoted2zexpr}: PVALUE \rightarrow Z\_EXPR;$
$\boldsymbol{unquoted2zexpr}: PVALUE \rightarrow Z\_EXPR$

$\forall\ pv: PVALUE \bullet$
$quoted2zexpr\ pv = PvalueZexpr\ (quote\_pval\ pv)$
$\land \quad unquoted2zexpr\ pv = PvalueZexpr\ (strip\_pval\ pv)$

A scalar literal is translated into Z using the following functions:

z

$\boldsymbol{lscalar2zexpr}: PLV \rightarrow Z\_EXPR$

$\forall\ PLV;\ subex1,\ subex2: Z\_EXPR$
$\mid psign = Positive \Rightarrow subex1 = PvalueZexpr\ power$
$\land\ psign = Negative \Rightarrow subex1 = Application\ (Ident\ INeg,\ PvalueZexpr\ power)$
$\land\ subex2 = ZInfixOps\ (PvalueZexpr\ value,\ \langle(Ei,\ subex1)\rangle)$
$\bullet\ sign = Positive \Rightarrow lscalar2zexpr\ (\theta PLV) = subex2$
$\land\ sign = Negative \Rightarrow lscalar2zexpr\ (\theta PLV) = Application\ (Ident\ RNeg,\ subex2)$

4.3.2 Operators and Functions

Operators and functions are translated from matlab/fcn expressions to Z using the following maps:

z

$$\boldsymbol{fcninfix2zexpr}: seq\ CHAR \twoheadrightarrow PVALUE$$

$$\begin{array}{lll}
fcninfix2zexpr = \{ & & \\
\texttt{"+"} \mapsto +_f, & \texttt{"-"} \mapsto -_f, & \\
\texttt{"*"} \mapsto *_f, & \texttt{"/"} \mapsto /_f, & \texttt{"\^{}"} \mapsto \hat{}_f, \\
\texttt{"="} \mapsto =_f, & \texttt{"!="} \mapsto \neq_f, & \\
\texttt{">"} \mapsto >_f, & \texttt{">="} \mapsto >=_f, & \\
\texttt{"<"} \mapsto <_f, & \texttt{"<="} \mapsto <=_f, & \\
\texttt{"\&\&"} \mapsto and_f, & \texttt{"||"} \mapsto or_f\} &
\end{array}$$

z

$$\boldsymbol{matinfix2zexpr}: seq\ CHAR \twoheadrightarrow PVALUE$$

$$\begin{array}{lll}
matinfix2zexpr = \{ & & \\
\texttt{"+"} \mapsto .+_m, & \texttt{".+"} \mapsto .+_m, & \texttt{"-"} \mapsto .-_m, \\
\texttt{".-"} \mapsto .-_m, & \texttt{"*"} \mapsto .*_m, & \texttt{".*"} \mapsto .*_m, \\
\texttt{"/"} \mapsto ./_m, & \texttt{"./"} \mapsto ./_m, & \texttt{"\textbackslash\textbackslash"} \mapsto .\backslash_m, \\
\texttt{".\textbackslash\textbackslash"} \mapsto .\backslash_m, & \texttt{"=="} \mapsto ==_m, & \\
\texttt{"\textasciitilde="} \mapsto \sim=_m, & \texttt{">"} \mapsto >_m, & \texttt{">="} \mapsto >=_m, \\
\texttt{"<"} \mapsto <_m, & \texttt{"<="} \mapsto <=_m, & \texttt{"\&"} \mapsto and_m, \\
\texttt{"|"} \mapsto or_m, & \texttt{"\^{}"} \mapsto .\hat{}_m, & \texttt{".\^{}"} \mapsto .\hat{}_m\}
\end{array}$$

z

$$\boldsymbol{fcnunary2zexpr}: seq\ CHAR \twoheadrightarrow PVALUE$$

$$\begin{array}{l}
fcnunary2zexpr = \{ \\
\texttt{"+"} \mapsto mp_f,\ \texttt{"-"} \mapsto mm_f,\ \texttt{"!"} \mapsto not_f\}
\end{array}$$

z

$$\boldsymbol{matunary2zexpr}: seq\ CHAR \twoheadrightarrow PVALUE$$

$$\begin{array}{l}
matunary2zexpr = \{ \\
\texttt{"+"} \mapsto mp_m,\ \texttt{"-"} \mapsto mm_m, \texttt{"\textasciitilde"} \mapsto \sim_m\}
\end{array}$$

It is intended that the above mappings generate distinct unused Z identifiers which will be defined in the ClawZ Z library with the correct semantic, fixity and precedence. The mappings for fcn expressions and matlab expressions are distinct because the precedences may differ. They are used for prefix and infix operator names, and for function names. It is expected that these mappings are applied by the parser, the appropriate mapping being chosen according to whether an Fcn or a Matlab expression is being parsed, so that the resulting parse tree has the correct identifiers for use in the Z.

4.3.3 Matlab Names

Matlab names ocurring in expressions may be global variables defined using a .m file, or local variables introduced by a masked subsystem. It is required that the global variables be subject to a systematic transformation controlled by the CLawZ user since in some applications exactly the same names are used elsewhere. However, it is less desirable that such transformations take place on the local variables.

The names changes are therefore implemented as a separate pass over the parameter between parsing and translation into Z. This is also used to obtain information about which of the local variables are actually used in parameters, which is helpful in minimising the amount of information passed to subsystems or library block invocations.

The required transformation therefore takes as a parameter and returns as a result a list of matlab names. When passed as parameter the list is the complete list of variables which are local in the current context. When returned the list is the set of local variables which occur in the parameter. In the transformed parameter, all global matlab names have been transformed as specified by the ClawZ user.

z

$$\begin{array}{l}
\boldsymbol{change\_atexp\_mlnames}: \mathbb{F}\ PVALUE \to AT\_EXP \to AT\_EXP \times \mathbb{F}\ PVALUE;\\
\boldsymbol{change\_sexp\_mlnames}: \mathbb{F}\ PVALUE \to SEXP \to SEXP \times \mathbb{F}\ PVALUE
\end{array}$$

$$\begin{array}{l}
\forall lvs: \mathbb{F}\ PVALUE;\ plv: PLV;\ sexp, sexp2: SEXP;\ pv: PVALUE;\\
\quad atexp: AT\_EXP;\ ssexp: seq\ SEXP;\ spvatexp: seq\ (PVALUE \times AT\_EXP) \bullet\\
\qquad change\_atexp\_mlnames\ lvs\ (Lscalar\ plv) = (Lscalar\ plv, \{\})\\
\land \qquad change\_atexp\_mlnames\ lvs\ (Brackets\ sexp) =\\
\qquad\qquad (\mu\ sexp': SEXP;\ ulvs: \mathbb{F}\ PVALUE\\
\qquad\qquad \mid (sexp', ulvs) = change\_sexp\_mlnames\ lvs\ sexp\\
\qquad\qquad \bullet (Brackets\ sexp', ulvs))\\
\land \qquad change\_atexp\_mlnames\ lvs\ (PrefixOp\ (pv, atexp)) =\\
\qquad\qquad (\mu\ atexp': AT\_EXP;\ ulvs: \mathbb{F}\ PVALUE\\
\qquad\qquad \mid (atexp', ulvs) = change\_atexp\_mlnames\ lvs\ atexp\\
\qquad\qquad \bullet (PrefixOp\ (pv, atexp'), ulvs))\\
\land \qquad change\_atexp\_mlnames\ lvs\ (Function\ (pv, sexp)) =\\
\qquad\qquad (\mu\ sexp': SEXP;\ ulvs: \mathbb{F}\ PVALUE\\
\qquad\qquad \mid (sexp', ulvs) = change\_sexp\_mlnames\ lvs\ sexp\\
\qquad\qquad \bullet (Function\ (pv, sexp'), ulvs))\\
\land \qquad change\_atexp\_mlnames\ lvs\ (Function2\ (pv, sexp, sexp2)) =\\
\qquad\qquad (\mu\ sexp', sexp2': SEXP;\ ulvs, ulvs2: \mathbb{F}\ PVALUE\\
\qquad\qquad \mid (sexp', ulvs) = change\_sexp\_mlnames\ lvs\ sexp\\
\qquad\qquad \land (sexp2', ulvs2) = change\_sexp\_mlnames\ lvs\ sexp2\\
\qquad\qquad \bullet (Function2\ (pv, sexp', sexp2'), ulvs \cup ulvs2))\\
\land \qquad change\_atexp\_mlnames\ lvs\ (Mlname\ (pv, ssexp)) =\\
\qquad\qquad (\mu\ ssexpulvs': seq\ (SEXP \times \mathbb{F}\ PVALUE);\ ulvs: \mathbb{F}\ PVALUE;\\
\qquad\qquad\qquad ssexp': seq\ SEXP\\
\qquad\qquad \mid ssexpulvs' = ssexp \fatsemi (change\_sexp\_mlnames\ lvs)\\
\qquad\qquad \land ulvs = \bigcup\ (ran(ssexpulvs' \fatsemi second))\\
\qquad\qquad \land ssexp' = ssexpulvs' \fatsemi first\\
\qquad\qquad \bullet \qquad if\ \ pv \in lvs\\
\qquad\qquad\qquad then\ (Mlname\ (pv, ssexp'), ulvs \cup \{pv\})\\
\qquad\qquad\qquad else\ (Mlname\ (change\_mlname\ pv, ssexp'), ulvs))\\
\land \qquad change\_sexp\_mlnames\ lvs\ (Sexpres\ (atexp, spvatexp)) =\\
\qquad\qquad (\mu\ ulvs1, ulvs2: \mathbb{F}\ PVALUE;\ saeulvs: seq\ (AT\_EXP \times \mathbb{F}\ PVALUE);\\
\qquad\qquad\quad atexp': AT\_EXP;\ spvatexp': seq\ (PVALUE \times AT\_EXP)\\
\qquad\qquad \mid (atexp', ulvs1) = change\_atexp\_mlnames\ lvs\ atexp\\
\qquad\qquad \land saeulvs = spvatexp \fatsemi (\lambda x{:}\mathbb{U};y{:}\mathbb{U} \bullet\ change\_atexp\_mlnames\ lvs\ y)\\
\qquad\qquad \land ulvs2 = \bigcup\ (ran(saeulvs \fatsemi (\lambda x{:}\mathbb{U};y{:}\mathbb{U} \bullet\ y)))\\
\qquad\qquad \land spvatexp' = (\lambda n{:}dom(spvatexp) \bullet\ ((spvatexp\ n).1, (saeulvs\ n).1))\\
\qquad\qquad \bullet (Sexpres\ (atexp', spvatexp'), ulvs1 \cup ulvs2))
\end{array}$$

z

$$\boldsymbol{change\_vexp\_mlnames}: \mathbb{F}\ PVALUE \to VEXP \to VEXP \times \mathbb{F}\ PVALUE$$

$$\begin{array}{l}
\forall lvs: \mathbb{F}\ PVALUE;\ plv: PLV;\ sexp,\ sexp2: SEXP;\ vexp: VEXP;\ pv: PVALUE;\\
\quad atexp: AT\_EXP;\ ssexp: seq\ SEXP;\ spvatexp: seq\ (PVALUE \times AT\_EXP)\bullet\\
\\
\quad change\_vexp\_mlnames\ lvs\ (LvDisplay\ ssexp) =\\
\qquad (\mu\ ssexpulvs': seq\ (SEXP \times \mathbb{F}\ PVALUE);\ ulvs: \mathbb{F}\ PVALUE;\\
\qquad\quad ssexp': seq\ SEXP\\
\qquad \mid ssexpulvs' = ssexp \fatsemi (change\_sexp\_mlnames\ lvs)\\
\qquad \land\ ulvs = \bigcup\ (ran(ssexpulvs' \fatsemi second))\\
\qquad \land\ ssexp' = ssexpulvs' \fatsemi first\\
\qquad \bullet\ (LvDisplay\ ssexp',\ ulvs))\\
\land \quad change\_vexp\_mlnames\ lvs\ (LvSlice\ (sexp,\ sexp2)) =\\
\qquad (\mu\ sexp',\ sexp2': SEXP;\ ulvs,\ ulvs2: \mathbb{F}\ PVALUE\\
\qquad \mid (sexp',\ ulvs) = change\_sexp\_mlnames\ lvs\ sexp\\
\qquad \land\ (sexp2',\ ulvs2) = change\_sexp\_mlnames\ lvs\ sexp2\\
\qquad \bullet\ (LvSlice\ (sexp',\ sexp2'),\ ulvs \cup ulvs2))\\
\land \quad change\_vexp\_mlnames\ lvs\ (LvArray\ (pv,\ vexp)) =\\
\qquad (\mu\ vexp': VEXP;\ ulvs: \mathbb{F}\ PVALUE\\
\qquad \mid (vexp',\ ulvs) = change\_vexp\_mlnames\ lvs\ vexp\\
\qquad \bullet \quad if\ pv \in lvs\\
\qquad\quad then\ (LvArray\ (pv,\ vexp'),\ ulvs \cup \{pv\})\\
\qquad\quad else\ (LvArray\ (change\_mlname\ pv,\ vexp'),\ ulvs))
\end{array}$$

z

$$\boldsymbol{change\_mexp\_mlnames}: \mathbb{F}\ PVALUE \to MEXP \to MEXP \times \mathbb{F}\ PVALUE$$

$$\begin{array}{l}
\forall lvs: \mathbb{F}\ PVALUE;\ plv: PLV;\ svexp: seq\ VEXP;\ pv: PVALUE;\\
\quad atexp: AT\_EXP;\ ssexp: seq\ SEXP;\ spvatexp: seq\ (PVALUE \times AT\_EXP)\bullet\\
\\
\quad change\_mexp\_mlnames\ lvs\ svexp =\\
\qquad (\mu\ svexpulvs': seq\ (VEXP \times \mathbb{F}\ PVALUE);\ ulvs: \mathbb{F}\ PVALUE;\\
\qquad\quad svexp': seq\ VEXP\\
\qquad \mid svexpulvs' = svexp \fatsemi (change\_vexp\_mlnames\ lvs)\\
\qquad \land\ ulvs = \bigcup\ (ran(svexpulvs' \fatsemi second))\\
\qquad \land\ svexp' = svexpulvs' \fatsemi first\\
\qquad \bullet\ (svexp',\ ulvs))
\end{array}$$

z

$$
\begin{array}{|l}
\quad \boldsymbol{change\_parameter\_mlnames}: \mathbb{F}\ PVALUE \rightarrow PARAMETER \\
\qquad\quad \rightarrow PARAMETER \times \mathbb{F}\ PVALUE \\
\hline
\quad \forall lvs: \mathbb{F}\ PVALUE; pv: PVALUE; sexp: SEXP; vexp: VEXP; mexp: MEXP \bullet \\
\\
\quad change\_parameter\_mlnames\ lvs\ (ParName\ pv) = \\
\qquad\quad if\ pv \in lvs \\
\qquad\quad then\ (ParName\ pv, \{pv\}) \\
\qquad\quad else\ (ParName\ (change\_mlname\ pv), \{\}) \\
\land \quad change\_parameter\_mlnames\ lvs\ (ParScalar\ sexp) = \\
\qquad\quad (\mu ulvs: \mathbb{F}\ PVALUE; sexp': SEXP \\
\qquad\quad \mid (sexp', ulvs) = change\_sexp\_mlnames\ lvs\ sexp \\
\qquad\quad \bullet (ParScalar\ sexp', ulvs)) \\
\land \quad change\_parameter\_mlnames\ lvs\ (ParVector\ vexp) = \\
\qquad\quad (\mu ulvs: \mathbb{F}\ PVALUE; vexp': VEXP \\
\qquad\quad \mid (vexp', ulvs) = change\_vexp\_mlnames\ lvs\ vexp \\
\qquad\quad \bullet (ParVector\ vexp', ulvs)) \\
\land \quad change\_parameter\_mlnames\ lvs\ (ParMatrix\ mexp) = \\
\qquad\quad (\mu ulvs: \mathbb{F}\ PVALUE; mexp': MEXP \\
\qquad\quad \mid (mexp', ulvs) = change\_mexp\_mlnames\ lvs\ mexp \\
\qquad\quad \bullet (ParMatrix\ mexp', ulvs))
\end{array}
$$

4.3.4 Expressions

z

$$
\begin{array}{|l}
\textbf{atscalar2zexpr}: AT\_EXP \rightarrow Z\_EXPR;\\
\textbf{scalar2zexpr}: SEXP \rightarrow Z\_EXPR\\
\hline
\forall plv: PLV;\ se1, se2: SEXP;\ pv: PVALUE;\ ae: AT\_EXP;\\
\quad spvae: seq\ (PVALUE \times AT\_EXP) \bullet\\
atscalar2zexpr\ (Lscalar\ plv) = ZBrackets\ (lscalar2zexpr\ plv)\\
\land\ atscalar2zexpr\ (Brackets\ se1) = ZBrackets\ (scalar2zexpr\ se1)\\
\land\ atscalar2zexpr\ (PrefixOp\ (pv, ae))\\
= Application\ (PvalueZexpr\ pv,\ ZBrackets\ (atscalar2zexpr\ ae))\\
\land\ atscalar2zexpr\ (Function\ (pv, se1))\\
= Application\ (PvalueZexpr\ pv,\ ZBrackets\ (scalar2zexpr\ se1))\\
\land\ atscalar2zexpr\ (Function2\ (pv, se1, se2))\\
= Application\ (PvalueZexpr\ pv,\ ZPair\ (scalar2zexpr\ se1,\ scalar2zexpr\ se2))\\
\land\ atscalar2zexpr\ (Mlname\ (pv,\ \langle\rangle))\\
= Ident\ (pvalue2ident\ pv)\\
\land\ atscalar2zexpr\ (Mlname\ (pv,\ \langle se1\rangle))\\
= Application\ (Ident\ (pvalue2ident\ pv),\\
\quad ZBrackets(Application\ (Ident\ R2zi,\ ZBrackets(scalar2zexpr\ se1))))\\
\land\ atscalar2zexpr\ (Mlname\ (pv, \langle se1, se2\rangle))\\
= Application(\\
\quad Application\ (\\
\quad\quad Ident\ (pvalue2ident\ pv),\\
\quad\quad ZBrackets(Application\ (Ident\ R2zi,\ ZBrackets(scalar2zexpr\ se1)))),\\
\quad ZBrackets(Application\ (Ident\ R2zi,\ ZBrackets(scalar2zexpr\ se2))))\\
\land\ scalar2zexpr\ (Sexpres\ (ae, spvae)) = ZInfixOps\ (atscalar2zexpr\ ae,\ spvae\ \fatsemi\\
\ (\lambda x{:}\mathbb{U} \bullet\ (pvalue2ident\ (first\ x),\ atscalar2zexpr\ (second\ x))))
\end{array}
$$

z

$$
\begin{array}{|l}
\textbf{vector2zexpr}: VEXP \rightarrow Z\_EXPR\\
\hline
\forall row: seq\ SEXP;\ se1, se2: SEXP;\ pv: PVALUE;\ ve: VEXP \bullet\\
vector2zexpr\ (LvDisplay\ row) = ZSequence\ (row\ \fatsemi\ scalar2zexpr)\\
\land\ vector2zexpr\ (LvSlice\ (se1, se2))\\
= ZInfixOps\ (scalar2zexpr\ se1,\ \langle(Slicei,\ scalar2zexpr\ se2)\rangle)\\
\land\ vector2zexpr\ (LvArray\ (pv,\ ve))\\
= ZInfixOps\ (Ident\ (pvalue2ident\ pv),\\
\quad \langle(Composei,\ Ident\ R2zi),\\
\quad (Composei,\ ZBrackets(vector2zexpr\ ve))\rangle)
\end{array}
$$

z

$$\boldsymbol{matrix2zexpr}: MEXP \rightarrow Z\_EXPR$$

$$\forall me: MEXP \bullet$$
$$matrix2zexpr\ me = ZSequence\ (((me \upharpoonright (\mathbb{U} \setminus \{LvDisplay\ \langle\rangle\})) \fatsemi vector2zexpr)$$

z

$$\boldsymbol{fcn2zexpr}: SEXP \rightarrow Z\_EXPR$$

$$\forall se: SEXP \bullet$$
$$fcn2zexpr\ se = ZLambdauU\ (scalar2zexpr\ se)$$

4.4 Parameter Evaluation

For certain purposes evaluation of parameter expressions is desirable.

It is needed:

- in order to determine the dimensions of arrays which are needed for type inference.
- in order to permit expression simplification during virtualization of some Simulink blocktypes (e.g. *Selector*).

First a function which gives the power of a *PLV* as an integer.

z

$$\boldsymbol{plv\_power}: PLV \rightarrow \mathbb{Z}$$

$$\forall plv: PLV \bullet$$
$$plv\_power\ plv =$$
$$\quad if\ plv.psign = Positive$$
$$\quad then\ (num2pvalue^{\sim})\ plv.power$$
$$\quad else\ \sim ((num2pvalue^{\sim})plv.power)$$

Then one which determines the value of a PLV.

z

$$\boldsymbol{plv\_val}: PLV \rightarrow \mathbb{R}$$

$$\forall plv: PLV \bullet$$
$$plv\_val\ plv = ($$
$$\quad \mu\ r: \mathbb{R}$$
$$\quad \mid r = (real\ ((num2pvalue^{\sim})\ plv.value)) *_R ((real\ 10)\ \hat{}_Z\ (plv\_power\ plv))$$
$$\quad \bullet\ if\ plv.sign = Positive\ then\ r\ else\ \sim_R r)$$

Next functions which attempt to determine the value of an atomic or scalar expressions. They return a value only if the expression is an integer literal, possibly with some brackets.

z

$$
\begin{array}{l}
\boldsymbol{atexp\_val}: AT\_EXP \rightarrowtail\!\!\!\!\!\!+\; \mathbb{R};\\
\boldsymbol{sexp\_val}: SEXP \rightarrowtail\!\!\!\!\!\!+\; \mathbb{R}\\
\hline
(\forall atexp: AT\_EXP;\ val: \mathbb{R} \bullet\\
atexp \mapsto val \in atexp\_val\\
\Leftrightarrow\\
\quad (\exists\ plv: PLV \bullet\ atexp = Lscalar\ plv \land val = plv\_val\ plv)\\
\lor\ (\exists\ sexp: SEXP \bullet\\
\qquad atexp = Brackets\ sexp\\
\qquad \land\ sexp \mapsto val \in sexp\_val));\\
\\
(\forall sexp: SEXP;\ val: \mathbb{R} \bullet\\
sexp \mapsto val \in sexp\_val\\
\Leftrightarrow (\exists\ atexp: AT\_EXP \bullet\\
\qquad sexp = Sexpres\ (atexp,\ \langle\rangle)\\
\qquad \land\ atexp \mapsto val \in atexp\_val))
\end{array}
$$

Finally (for *Selector*) we partially specify the value of vector expressions. Note that this is in fact used only for indexing and therefore the integers must be represented exactly and others need not be represented at all (in an implementation).

z

$$
\begin{array}{l}
\boldsymbol{vexp\_val}: VEXP \rightarrowtail\!\!\!\!\!\!+\; seq\ \mathbb{R}\\
\hline
\forall vexp: VEXP;\ val: seq\ \mathbb{R} \bullet\\
\quad vexp \mapsto val \in vexp\_val\\
\quad \Leftrightarrow\\
\qquad (\exists\ seqsexp: seq\ SEXP \bullet\\
\qquad\quad \#val = \#seqsexp\\
\qquad \land\ vexp = LvDisplay\ seqsexp\\
\qquad \land\ val = seqsexp \mathbin{⨾} sexp\_val)\\
\quad \lor\\
\qquad (\exists\ le,\ re: SEXP;\ lv,\ rv: \mathbb{R};\ li,\ ri: \mathbb{Z}\\
\qquad \mid LvSlice\ (le,\ re) = vexp\\
\qquad \land\ le \mapsto lv \in sexp\_val\\
\qquad \land\ re \mapsto rv \in sexp\_val\\
\qquad \bullet\ val =\\
\qquad\qquad \{n: \mathbb{N}\\
\qquad\qquad \mid real\ n \leq_R rv -_R lv\\
\qquad\qquad \bullet\ (n+1) \mapsto lv +_R (real\ n)\})
\end{array}
$$

4.5 Parameter Typing

First we provide a partial specification of the length of vector expressions:

z

$$
\begin{array}{l}
\boldsymbol{vexp\_length}: VEXP \rightarrowtail\!\!\!\!\!\!\;\;\, \mathbb{N} \\
\hline
\forall vexp: VEXP;\ length: \mathbb{N} \bullet \\
vexp \mapsto length \in vexp\_length \\
\Leftrightarrow \\
\quad (\exists\ ssexp: seq\ SEXP \bullet \\
\qquad\quad vexp = LvDisplay\ ssexp \\
\qquad\quad \wedge\ length = \#ssexp) \\
\vee\ (\exists\ sexp1,\ sexp2: SEXP;\ lr,\ rr: \mathbb{R} \bullet \\
\qquad\quad vexp = LvSlice\ (sexp1,\ sexp2) \\
\qquad\quad \wedge\ sexp1 \mapsto lr \in sexp\_val \\
\qquad\quad \wedge\ sexp2 \mapsto rr \in sexp\_val \\
\qquad\quad \wedge\ real\ length = rr -_R lr +_R (real\ 1)) \\
\vee\ (\exists\ pv: PVALUE;\ vexp2: VEXP \bullet \\
\qquad\quad vexp = LvArray\ (pv,\ vexp2) \\
\qquad\quad \wedge\ vexp2 \mapsto length \in vexp\_length)
\end{array}
$$

Finally the dimensions of matrix expressions.

z

$$
\begin{array}{l}
\boldsymbol{mexp\_length}: MEXP \rightarrowtail\!\!\!\!\!\!\;\;\, seq\ \mathbb{N} \\
\hline
\forall mexp: MEXP;\ s: seq\ \mathbb{N} \bullet \\
mexp \mapsto s \in mexp\_length \\
\Leftrightarrow \\
(\exists d: \mathbb{N} \bullet\ s = \langle \#mexp,\ d \rangle\ \wedge \\
\{v{:}ran\ mexp;\ n: \mathbb{N} \mid v \mapsto n \in vexp\_length \bullet n\} = \{d\})
\end{array}
$$

We now define a function which determines the type of a *PARAMETER*. The function is parameterised by a *MVARTYPES* which it refers to if the parameter is simply a variable.

z

$$
\begin{array}{|l}
\quad \boldsymbol{parameter\_type}: MVARTYPES \rightarrow PARAMETER \nrightarrow MVARTYPE \\
\hline
\quad \forall mvts: MVARTYPES;\ mvt: MVARTYPE \bullet \\
\quad (\forall pv: PVALUE \bullet \\
\qquad ParName\ pv \mapsto mvt \in parameter\_type\ mvts \\
\qquad \Leftrightarrow pv \mapsto mvt \in mvts) \\
\land \quad (\forall se: SEXP \bullet \\
\qquad ParScalar\ se \mapsto mvt \in parameter\_type\ mvts \\
\qquad \Leftrightarrow mvt = \langle\rangle) \\
\land \quad (\forall ve: VEXP;\ l: \mathbb{N} \bullet \\
\qquad ParVector\ ve \mapsto mvt \in parameter\_type\ mvts \\
\qquad \Leftrightarrow ve \mapsto l \in vexp\_length \land mvt = \langle l \rangle \\
\qquad \lor ve \notin dom\ vexp\_length \land mvt = \langle 0 \rangle) \\
\land \quad (\forall me: MEXP \bullet \\
\qquad ParMatrix\ me \mapsto mvt \in parameter\_type\ mvts \\
\qquad \Leftrightarrow me \mapsto mvt \in mexp\_length \\
\qquad \lor me \notin dom\ mexp\_length \land mvt = \langle \# me, 0 \rangle)
\end{array}
$$

The following gives the length of a variable from its type, or zero if it cannot be determined.

z

$$
\begin{array}{|l}
\quad \boldsymbol{mvt\_length}: MVARTYPE \rightarrow \mathbb{N} \\
\hline
\quad mvt\_length\ \langle\rangle = 1; \\
\quad \forall\ mvt: MVARTYPE;\ n: \mathbb{N} \bullet \\
\quad mvt\_length\ (mvt \frown \langle n \rangle) = n * mvt\_length\ mvt
\end{array}
$$

4.6 Parameter Translation

Parameter translation methods are selected according to the translation type of the parameter as specified in the transmitted parameter information in the metadata.

This is translated into abstract Z according to the following specifications.

z

$$\begin{array}{l}
\boldsymbol{scalar\_par\_trans}: PARAMETER \rightarrow TRANSLATION\_RESULT \\
\hline
\forall pn: PVALUE; se: SEXP; ve: VEXP; me: MEXP \bullet \\
\quad scalar\_par\_trans\ (ParName\ pn) = TMatch\ (Ident\ (pvalue2ident\ pn)) \\
\land \quad scalar\_par\_trans\ (ParScalar\ se) = TMatch\ (scalar2zexpr\ se) \\
\land \quad scalar\_par\_trans\ (ParVector\ ve) = TNoMatch \\
\land \quad scalar\_par\_trans\ (ParMatrix\ me) = TNoMatch
\end{array}$$

z

$$\begin{array}{l}
\boldsymbol{vector\_par\_trans}: PARAMETER \rightarrow TRANSLATION\_RESULT \\
\hline
\forall pn: PVALUE; se: SEXP; ve: VEXP; me: MEXP \bullet \\
\quad vector\_par\_trans(ParName\ pn) = TMatch(Ident\ (pvalue2ident\ pn)) \\
\land \quad vector\_par\_trans(ParScalar\ se) = TMatch(ZSequence\ \langle scalar2zexpr\ se\rangle) \\
\land \quad vector\_par\_trans(ParVector\ ve) = TMatch(vector2zexpr\ ve) \\
\land \quad vector\_par\_trans(ParMatrix\ me) = TNoMatch
\end{array}$$

z

$$\begin{array}{l}
\boldsymbol{matrix\_par\_trans}: PARAMETER \rightarrow TRANSLATION\_RESULT \\
\hline
\forall pn: PVALUE; se: SEXP; ve: VEXP; me: MEXP \bullet \\
\quad matrix\_par\_trans\ (ParName\ pn) \\
\quad = TMatch\ (Ident\ (pvalue2ident\ pn)) \\
\land \quad matrix\_par\_trans\ (ParScalar\ se) \\
\quad = TMatch\ (ZSequence\ \langle ZSequence\ \langle scalar2zexpr\ se\rangle\rangle) \\
\land \quad matrix\_par\_trans\ (ParVector\ ve) \\
\quad = TMatch\ (ZSequence\ \langle vector2zexpr\ ve\rangle) \\
\land \quad matrix\_par\_trans\ (ParMatrix\ me) \\
\quad = TMatch\ (matrix2zexpr\ me)
\end{array}$$

z

$$\begin{array}{l}
\boldsymbol{SVM\_par\_trans}: PARAMETER \rightarrow TRANSLATION\_RESULT \\
\hline
\forall pn: PVALUE; se: SEXP; ve: VEXP; me: MEXP \bullet \\
\quad SVM\_par\_trans\ (ParName\ pn) = scalar\_par\_trans\ (ParName\ pn) \\
\land \quad SVM\_par\_trans\ (ParScalar\ se) = scalar\_par\_trans\ (ParScalar\ se) \\
\land \quad SVM\_par\_trans\ (ParVector\ ve) = vector\_par\_trans\ (ParVector\ ve) \\
\land \quad SVM\_par\_trans\ (ParMatrix\ me) = matrix\_par\_trans\ (ParMatrix\ me)
\end{array}$$

z

```
unified_par_trans: PARAMETER → TRANSLATION_RESULT
──────────────────────────────
 ∀ pn: PVALUE; se: SEXP; ve: VEXP; me: MEXP •
 unified_par_trans (ParName pn)
 = TMatch (Ident (pvalue2ident pn))
∧ unified_par_trans (ParScalar se)
 = TMatch (Application (Ident S2Ui, ZBrackets(scalar2zexpr se)))
∧ unified_par_trans (ParVector ve)
 = TMatch (Application (Ident V2Ui, ZBrackets(vector2zexpr ve)))
∧ unified_par_trans (ParMatrix me)
 = TMatch (Application (Ident M2Ui, ZBrackets(matrix2zexpr me)))
```

z

```
checkbox_par_trans: PVALUE → TRANSLATION_RESULT
──────────────────────────────
 ∀ pv: PVALUE •
 checkbox_par_trans pv =
 if pv = off then TMatch (PvalueZexpr (sc2pv "real 0"))
 else if pv = on then TMatch (PvalueZexpr (sc2pv "real 1"))
 else TNoMatch
```

z

```
popup_par_trans: seq PVALUE → PVALUE → TRANSLATION_RESULT
──────────────────────────────
 ∀ spv: seq PVALUE; pv: PVALUE •
 popup_par_trans spv pv =
 if pv ∈ ran spv
 then
         (μn:ℕ
         | spv n = pv
         • TMatch (Application (PvalueZexpr (sc2pv "real"), ZNat n)))
 else TNoMatch
```

The following specifies how a mask parameter is to be translated and is for use when translating masked subsystems. Its parameters are the relevant mask style, the parameter and the set of local names in scope. It returns a *Z_EXPR* and the set of local names which occurred in the parameter. Global but not local names are subject to the user specified transformation.

z

$$\begin{array}{l}
\boldsymbol{maskparam\_trans}: MASK\_STYLE \times PVALUE \times \mathbb{F}\ PVALUE \\
\qquad \nrightarrow TRANSLATION\_RESULT \times \mathbb{F}\ PVALUE
\end{array}$$

$$\begin{array}{l}
\forall style: MASK\_STYLE;\ pv: PVALUE;\ freevars: \mathbb{F}\ PVALUE;\ spv: seq\ PVALUE; \\
\quad tr: TRANSLATION\_RESULT;\ used\_maskvars: \mathbb{F}\ PVALUE \bullet \\
\quad (style, pv, freevars) \mapsto (tr, used\_maskvars) \in maskparam\_trans \\
\Leftrightarrow \\
\quad style = MSEdit \\
\quad \land (\exists\ par, par': PARAMETER;\ ulvs: \mathbb{F}\ PVALUE;\ tr: TRANSLATION\_RESULT \\
\qquad \bullet\ pv \mapsto par \in parse\_param \\
\qquad \land (par', used\_maskvars) = change\_parameter\_mlnames\ freevars\ par \\
\qquad \land tr = SVM\_par\_trans\ par') \\
\\
\lor \quad style = MSCheckbox \\
\quad \land tr = checkbox\_par\_trans\ pv \\
\quad \land used\_maskvars = \{\} \\
\\
\lor \quad style = MSPopup\ spv \\
\quad \land tr = popup\_par\_trans\ spv\ pv \\
\quad \land used\_maskvars = \{\}
\end{array}$$

Now we put these together to specify parameter translation. This first version of the parameter translation function accepts the set of local (masked) variables in the context of the parameter, as a parameter, and returns the set of local variables which are used in the parameter. It transforms global matlab names as specified by the relevant user controls, but leaves the local variables unchanged.

z

$$
\begin{array}{l}
\textbf{par\_trans2}: \mathbb{F}\ PVALUE \to PVALUE \to PVALUE \\
\quad \to TRANSLATION\_RESULT \times \mathbb{F}\ PVALUE \\
\hline
\forall type, value: PVALUE; param, param': PARAMETER; \\
\quad lvs, ulvs: \mathbb{F}\ PVALUE; se: SEXP; popuplist: seq\ PVALUE \bullet \\
\quad par\_trans2\ lvs\ Quoted\ value = (TMatch\ (quoted2zexpr\ value), \{\}) \\
\land \quad par\_trans2\ lvs\ Unquoted\ value = (TMatch\ (unquoted2zexpr\ value), \{\}) \\
\land \quad par\_trans2\ lvs\ Fcn\ value = \\
\qquad if\ value \mapsto se \in parse\_fcn\_param \\
\qquad then\quad (\mu\ se': SEXP \\
\qquad\qquad |\ (se', ulvs) = change\_sexp\_mlnames\ (lvs \cup \{sc2pv\ \texttt{"u"}\})\ se \\
\qquad\qquad \bullet (TMatch\ (fcn2zexpr\ se'), ulvs)) \\
\qquad else\ (TNoMatch, \{\}) \\
\land \quad par\_trans2\ lvs\ Checkbox\ value = (checkbox\_par\_trans\ value, \{\}) \\
\land \quad (value \mapsto param \in parse\_param \\
\quad \Rightarrow \qquad (param', ulvs) = change\_parameter\_mlnames\ lvs\ param \\
\quad \land \qquad par\_trans2\ lvs\ Scalar\ value = (scalar\_par\_trans\ param', ulvs) \\
\quad \land \qquad par\_trans2\ lvs\ Vector\ value = (vector\_par\_trans\ param', ulvs) \\
\quad \land \qquad par\_trans2\ lvs\ Matrix\ value = (matrix\_par\_trans\ param', ulvs) \\
\quad \land \qquad par\_trans2\ lvs\ SVM\ value = (SVM\_par\_trans\ param', ulvs) \\
\quad \land \qquad par\_trans2\ lvs\ Unified\ value = (unified\_par\_trans\ param', ulvs)) \\
\land \quad (type \mapsto popuplist \in parse\_popup\_tc \\
\quad \Rightarrow par\_trans2\ lvs\ type\ value = (popup\_par\_trans\ popuplist\ value, \{\})) \\
\land \quad (type \notin \{Quoted, Unquoted, Fcn, Checkbox\} \\
\quad \land\ value \notin (dom(parse\_param) \cup dom(parse\_popup\_tc)) \\
\quad \Rightarrow par\_trans2\ lvs\ type\ value = (TNoMatch, \{\}))
\end{array}
$$

The following parameter translation method was specially devised for the Selector block but is included here in case it should prove useful for other block types.

The special characteristics of this parameter translation method are as follows:

- It is intended to translate a parameter which may be either a scalar or a vector and to make it easy to see in the result which of those two was obtained.
- It is required under some circumstances to treat a unit vector as if it were the scalar which is its sole element.
- When converting from unit vector to scalar it will strip off the outer brackets if it is a vector display, otherwise (some other vector expression which happens to have length 1) it will select the first element.
- In determining the dimensions of the parameter it makes use of the matlab variable type context.

- Parameters with dimensionality > 1 are rejected.
- It accepts a parameter which is the inferred port type for some output, this influences the treatment of the parameter as follows:
  - If the type is *UnknownPT* or *GenericPT* then a scalar result will be returned for a unit vector (or a scalar) parameter.
  - If the type is *ScalarPT* then a scalar result will be returned for a unit vector parameter, and a parameter which cannot be read as or coerced into a scalar will be rejected.
  - If the type is *VectorPT* or *BusPT* then a vector result will be returned.

The reasons for this specific behaviour are as follows:

- Selector blocks appear frequently to be used to select a single line using a parameter which is a unit vector display. In most of these cases the output will be connected to a block which we will have implemented with a scalar input.
- If the result of the translation is influenced by the result of signal type inference it will be possible for the user to specify in his steering file a type for the Selector output port (overriding signal type inference) in case the default interpretation is not correct (if he really wants a unit vector and would otherwise have got a scalar).

Since the specification is otherwise to large two auxiliary functions are specified which deal with the case that the parameter type is known and unknown respectively:

z

$$\mathbf{spt\_typed}: (MVARTYPE \times PORT\_TYPE \times PARAMETER) \\ \quad \to SPECIAL\_RESULT$$

$$\begin{array}{l}
\forall\ mvtype: MVARTYPE;\ port\_type: PORT\_TYPE;\ param: PARAMETER \bullet \\
spt\_typed\ (mvtype,\ port\_type,\ param) = \\
(\mu\ sr: SPECIAL\_RESULT;\ tr: TRANSLATION\_RESULT;\ ze: Z\_EXPR; \\
\quad bd: \mathbb{N} \times seq\ [line\_name: PVALUE;\ port\_type: PORT\_TYPE];\ n: \mathbb{N} \\
\mid \qquad \#mvtype = 0 \land tr = scalar\_par\_trans\ param \\
\qquad\quad \land\ (\ tr = TMatch\ ze \\
\qquad\qquad \land\ ((port\_type = ScalarPT \lor port\_type = GenericPT \\
\qquad\qquad\qquad \lor port\_type = UnknownPT) \\
\qquad\qquad\quad \land\ sr = SRScalar\ ze \\
\qquad\qquad\quad \lor\ (port\_type = VectorPT\ n \lor port\_type = BusPT\ bd) \\
\qquad\qquad\quad \land\ sr = SRVector\ (ZSequence\ \langle ze \rangle)) \\
\qquad\quad \lor\ (tr = TNoMatch \lor tr = TFail) \land sr = SRFail) \\
\qquad \lor\ \#mvtype = 1 \land tr = vector\_par\_trans\ param \\
\qquad\quad \land\ (\ tr = TMatch\ ze \\
\qquad\qquad \land\ (\ \ port\_type = ScalarPT \land mvtype\ 1 \in \{0,1\} \\
\qquad\qquad\quad \land\ sr = SRScalar\ (ZBrackets(Application\ (ze,\ ZNat\ 1))) \\
\qquad\qquad\quad \lor\ \ (port\_type = VectorPT\ n \lor port\_type = BusPT\ bd \\
\qquad\qquad\qquad \lor\ port\_type = GenericPT \lor port\_type = UnknownPT \\
\qquad\qquad\qquad \lor\ (port\_type = ScalarPT \land mvtype\ 1 > 1)) \\
\qquad\qquad\quad \land\ sr = SRVector\ ze) \\
\qquad\quad \lor\ (tr = TNoMatch \lor tr = TFail) \land sr = SRFail) \\
\qquad \lor\ \ \#mvtype > 1 \land sr = SRFail \\
\bullet\ sr)
\end{array}$$

z

$$
\begin{array}{l}
\quad \boldsymbol{spt\_untyped}: (PORT\_TYPE \times PARAMETER) \\
\qquad \rightarrow SPECIAL\_RESULT \\
\hline
\forall\ port\_type: PORT\_TYPE;\ param: PARAMETER \bullet \\
spt\_untyped\ (port\_type,\ param) = \\
(\mu\ sr: SPECIAL\_RESULT;\ tr: TRANSLATION\_RESULT;\ ze: Z\_EXPR; \\
\quad bd: \mathbb{N} \times seq\ [line\_name: PVALUE;\ port\_type: PORT\_TYPE];\ n: \mathbb{N} \\
|\quad (port\_type = ScalarPT \\
\qquad \wedge\ tr = scalar\_par\_trans\ param \\
\qquad \wedge\ \ (tr = TMatch\ ze \wedge sr = SRScalar\ ze \\
\qquad\qquad \vee\ (tr = TNoMatch \vee tr = TFail) \wedge sr = SRFail)) \\
\quad \vee\ ((port\_type = VectorPT\ n \vee port\_type = BusPT\ bd) \\
\qquad \wedge\ tr = vector\_par\_trans\ param \\
\qquad \wedge\ \ (tr = TMatch\ ze \wedge sr = SRVector\ ze \\
\qquad\qquad \vee\ (tr = TNoMatch \vee tr = TFail) \wedge sr = SRFail)) \\
\quad \vee\ ((port\_type = UnknownPT \vee port\_type = GenericPT) \\
\qquad \wedge\ sr = SRFail) \\
\bullet\ sr)
\end{array}
$$

This parameter translation method is needed both for synthesis and for virtualization. In the latter case more information is needed, since for optimisation it is necessary to be able to evaluate the parameter where possible. For this reason we first specify a version which returns a *PARAMETER* and then the original version which does not.

z

$$\textbf{special\_par\_trans\_wp}: MVARTYPES \rightarrow \mathbb{F}\ PVALUE \rightarrow PORT\_TYPE \rightarrow PVALUE \\ \rightarrow PARAMETER \times SPECIAL\_RESULT \times \mathbb{F}\ PVALUE$$

$\forall\ mvartypes: MVARTYPES;\ lvs: \mathbb{F}\ PVALUE;\ port\_type: PORT\_TYPE \bullet$

$special\_par\_trans\_wp\ mvartypes\ lvs\ port\_type =$

$(\lambda parpv: PVALUE \bullet ((\mu p{:}PARAMETER),\ SRFail,\ \{\}))$
$\oplus$
$\{parpv: PVALUE;\ param,\ param': PARAMETER;\ ulvs: \mathbb{F}\ PVALUE;$
$sr: SPECIAL\_RESULT;\ mvtype: MVARTYPE;\ tr: TRANSLATION\_RESULT;$
$ze: Z\_EXPR;\ bd: \mathbb{N} \times seq\ PORT\_DETAILS;\ n: \mathbb{N}$
$|\quad parpv \mapsto param \in parse\_param$
$\land\quad (param',\ ulvs) = change\_parameter\_mlnames\ lvs\ param$
$\land\quad (param' \mapsto mvtype \in parameter\_type\ mvartypes$
$\quad\quad \land\ sr = spt\_typed\ (mvtype,\ port\_type,\ param')$
$\quad \lor\ param' \notin dom(parameter\_type\ mvartypes)$
$\quad\quad \land\ sr = spt\_untyped\ (port\_type,\ param'))$
$\bullet\ parpv \mapsto (param',\ sr,\ ulvs)\}$

z

$$\textbf{special\_par\_trans}: MVARTYPES \rightarrow \mathbb{F}\ PVALUE \rightarrow PORT\_TYPE \rightarrow PVALUE \\ \rightarrow SPECIAL\_RESULT \times \mathbb{F}\ PVALUE$$

$\forall mvartypes: MVARTYPES;\ lvs: \mathbb{F}\ PVALUE;\ port\_type: PORT\_TYPE;\ parpv: PVALUE \bullet$
$special\_par\_trans\ mvartypes\ lvs\ port\_type\ parpv =$
$(\mu\ param: PARAMETER;\ ulvs: \mathbb{F}\ PVALUE;\ sr: SPECIAL\_RESULT$
$|\ (param,\ sr,\ ulvs) = special\_par\_trans\_wp\ mvartypes\ lvs\ port\_type\ parpv$
$\bullet\ (sr,\ ulvs))$

4.7 .m File Translation

The translation of Matlab .m files largely independent of the rest of the diagram translation. Some "type inference" is undertaken and the results are output in a form suitable for inclusion in a ClawZ steering file for a subsequent model translation.

A limited kind of Matlab .m file is supported consisting essentially of a series of equations on the left of which is a Matlab name and on the right an expression which would be acceptable to the Clawz translator as a "SVM" or "Unified" parameter value to a Simulink block.

These equations are translated into Z abbreviation definitions. The right hand side of the definitions are translated in exactly the same way as a Simulink block parameter, except that comments and line continuations are permitted.

We first define the type of the .m file translator.

The .m file translator is parameterised by a PVALUE which is a parameter type. This must be either "SVM" or "Unified", and determines how the expressions on the right of the equations are translated into Z. The manner in which this parameter is communicated to the .m file translator is left to the detailed design.

In addition to a Z specification consisting of the translation of the equations of the .m file into Z, the processor returns type information for the variables, which must be written to a file for use when translating Simulink models which use the variables defined in the .m file.

z

$$\boldsymbol{M\_FILE\_PROC} \mathrel{\widehat=} PVALUE \rightarrow M\_FILE \nrightarrow Z\_SPEC \times MVARTYPES$$

The translation of expressions in matlab .m files is a minor variant on their translation in Simulink block parameters.

The following *mexp_trans* specification is similar to *par_trans* above, differing the into following respects:

- Only translation codes *SVM* and *Unified* are accepted.
- The type of the expression (a *MVARTYPE*) is returned as well as the translation into Z. This is required for determining the type of the variable.
- It is assumed that there are no local variables (maskvars) in scope.

z

$$\begin{array}{l}
\boldsymbol{mexp\_trans}: PVALUE \rightarrow PVALUE \\
\qquad \nrightarrow TRANSLATION\_RESULT \times PARAMETER \\
\hline
\forall code, value: PVALUE;\ tr: TRANSLATION\_RESULT;\ param': PARAMETER \bullet \\
value \mapsto (tr, param') \in mexp\_trans\ code \\
\Leftrightarrow \\
(\exists param: PARAMETER;\ ulvs: \mathbb{F}\ PVALUE \bullet \\
\qquad value \mapsto param \in parse\_param \\
\land \qquad (param', ulvs) = change\_parameter\_mlnames\ \{\}\ param \\
\land \qquad (code = SVM \land tr = SVM\_par\_trans\ param' \\
\qquad \lor\ code = Unified \land tr = unified\_par\_trans\ param'))
\end{array}$$

We now define a parameter translator which updates a record of the types of the matlab variables.

z

$$
\begin{array}{l}
\boldsymbol{mexp\_trans2}: PVALUE \times PVALUE \times PVALUE \times MVARTYPES \\
\quad \rightarrow\!\!\!\!\!+ \; TRANSLATION\_RESULT \times MVARTYPES \\
\hline
\forall code, varname, expr: PVALUE; mvts, mvts': MVARTYPES; \\
\; tr: TRANSLATION\_RESULT \bullet \\
(code, varname, expr, mvts) \mapsto (tr, mvts') \in mexp\_trans2 \\
\Leftrightarrow (\exists param: PARAMETER; vartype: MVARTYPE \bullet \\
expr \mapsto (tr, param) \in mexp\_trans\ code \\
\land\ param \mapsto vartype \in parameter\_type\ mvts \\
\land\ mvts' = mvts \oplus \{change\_mlname\ varname \mapsto vartype\})
\end{array}
$$

The following auxiliary function constructs the abbreviation definitions to be output by the M file translator from the results of translating the right hand side of the equation as if it were a Simulink block parameter.

The function is partial and is not defined when the “translation result” is not a “TMatch”, i.e. where the translation of the right hand side of the equation fails. The required effect is that no Z paragraphs are output where translation of an equation fails. A diagnostic should be output, the details of which are left for the detailed design.

It should be noted that the M files may also contain (non-comment) statements which are not equations, which should be treated in the same way as an equation with an untranslatable right hand side, and in general, that the .m file translator should attempt to skip over anything which it does not understand and continue processing at the beginning of the next “logical line”. For a fuller statement on lexical and syntactic aspects of M file processing see section 2.3.

z

$$
\begin{array}{ll}
& \boldsymbol{make\_abbr\_def}: TRANSLATION\_RESULT \times PVALUE \rightarrow\!\!\!\!\!+ \; Z\_PARA \\
\hline
& dom\ (make\_abbr\_def) = \{z: Z\_EXPR; pv: PVALUE \bullet (TMatch\ z, pv)\} \\
\land & (\forall\ z\_expr: Z\_EXPR; mlname: PVALUE \bullet \\
& make\_abbr\_def\ (TMatch\ z\_expr, mlname) = \\
& \quad AbbrevDef\ (\\
& \qquad ident \mathrel{\widehat{=}} pvalue2ident\ (change\_mlname\ mlname), \\
& \qquad value \mathrel{\widehat{=}} z\_expr))
\end{array}
$$

Now we have a translation specification which delivers a Z spec and updated *MVARTYPES.* This is total since failing translations will deliver an empty specification. An equation is processed by processing the right hand side of the equation using *mexp_trans2* and then passing the resulting Z expression together with the name on the left of the equation to the *make_abbrev_def* function.

z

$$
\begin{array}{l}
\boldsymbol{mexp\_trans3}: PVALUE \times PVALUE \times PVALUE \times MVARTYPES \\
\quad \rightarrow Z\_SPEC \times MVARTYPES \\
\hline
\forall ptype, varname, expr: PVALUE; mvts, mvts': MVARTYPES; \\
\; ze: Z\_EXPR; zp: Z\_PARA; zs: Z\_SPEC \\
\mid ((ptype, varname, expr, mvts) \mapsto (TMatch\ ze, mvts') \in mexp\_trans2 \\
\quad \wedge zs = \langle make\_abbr\_def(TMatch\ ze, varname)\rangle) \\
\vee (\neg(\exists ze: Z\_EXPR; mvts': MVARTYPES \bullet \\
\qquad (ptype, varname, expr, mvts) \mapsto (TMatch\ ze, mvts') \in mexp\_trans2)) \\
\quad \wedge zs = \langle\rangle \wedge mvts' = mvts \\
\bullet \\
mexp\_trans3\ (ptype, varname, expr, mvts) = (zs, mvts')
\end{array}
$$

z

$$
\begin{array}{l}
\boldsymbol{m\_file\_proc}: M\_FILE\_PROC \\
\hline
\forall\ ptype: PVALUE \bullet \\
\quad m\_file\_proc\ ptype\ \langle\rangle = (\langle\rangle, \{\}); \\
\\
\forall\ ptype: PVALUE; m\_file: M\_FILE; mfeq: MFEQ; \\
\quad zs1, zs2: Z\_SPEC; mvts, mvts': MVARTYPES \\
\mid \quad m\_file\_proc\ ptype\ m\_file = (zs1, mvts) \\
\wedge \quad (zs2, mvts') = mexp\_trans3(ptype, change\_mlname\ mfeq.name, mfeq.value, mvts) \\
\bullet \quad m\_file\_proc\ ptype\ (m\_file \frown \langle mfeq\rangle) = (zs1 \frown zs2, mvts')
\end{array}
$$

5 INPUT FILE PROCESSING

This section defines various transformations of the information in the various input files to ClawZ which logically precede the main model translation.

5.1 Parsing of Structures

The ClawZ steering file has a syntactic structure similar to that of a Simulink model file and is specified in sections 2.1 and 2.2.

The following function should be understood to parse files of this structure. It must be understood to operate by side effect, since it accepts the name of a file which is to be read and parsed.

z

$$\boldsymbol{parse\_file}: STRING \rightarrow STRUCTURE$$

It is shown as a total function, failure to obtain a structure should cause the run of ClawZ to be aborted.

5.2 Matlab Variable Types

Type information is inferred and preserved about the type of variables set up in Matlab .m files. The information may then used when the variable is used as a parameter to a Simulink block. The following specification shows how the variable type information is extracted from the clawz steering file.

z

$$\boldsymbol{mvts\_of\_structure}: STRUCTURE \rightarrow MVARTYPES$$

$$\begin{array}{l}
\forall steerfile: STRUCTURE \bullet \\
mvts\_of\_structure\ steerfile = \\
\{pn: PNAME;\ pv1,\ pv2: PVALUE;\ m{:}MVARTYPE;\ struct: STRUCTURE; \\
\ \ snv,\ snvg: seq\ [name: PNAME;\ value: VALUE] \\
\mid Structure\ snvg = steerfile \\
\land\ (name \mathrel{\widehat=} VariableTypes,\ value \mathrel{\widehat=} Struct\ struct) \in ran\ snvg \\
\land\ Structure\ snv = struct \\
\land\ (name \mathrel{\widehat=} pn,\ value \mathrel{\widehat=} Simple\ (Expression,\ pv2)) \in ran\ snv \\
\land\ pv1 = sc2pv\ (pn2sc\ pn) \\
\land\ pv2 \mapsto m \in parse\_var\_type \\
\bullet\ pv1 \mapsto m\}
\end{array}$$

The structure is computed during .m file processing and then written to a file. It is read back as part of the steering file when translating a model which depends on the definitions in the .m file and then held as *mvartypes* and referred to as necessary during the model translation. At present this structure is only referred to during type inference for certain block-types.

6 MODEL TRANSLATION

The stages in the translation of a model are as follows:

1. The model is read and parsed to give a stucture of type *SYSTEM*
2. The steering file is read and parsed to give a structure of type *seq STEERSTRUCT*.
3. The model is transformed into a tree structure of type *A_BLOCK*.
4. Library look-up is undertaken on each of the non system/subsystem blocks in the system.
5. Port types set in the steering file are applied.

6. Signal Analysis is undertaken propagating information about port types and widths along lines and through bus constructors.
7. Block Synthesis is undertaken for certain kinds of blocks which have not been successfully matched against the library.
8. Subsystems are translated, generating a schema for each subsystem connecting the blocks in the subsystem using equations corresponding to the lines.
9. If a metadata output file has been named Metadata is generated for all subsystems of the model. The metadata will be written to the named file. This generated metadata becomes available too late to affect the translation, so when translating a library which contains internal block references it is necessary to run the translation a second time supplying the metadata created on the first run (and further runs may also be necessary if reference is made to subsystems which contain further internal block references to subsystems with free local variables).
10. The artificial subsystems are generated as additional structures of type *A_BLOCK*.

Specifications are then transcribed and filtered from the main system and artificial subsystems, first into *SYS_SPEC*s and then into output files (see section 7).

6.1 Model Transformation

In this section the parsed model is transformed into a data structure more convenient for analysis and specification generation.

First a general functions for extracting parameter values:

z

$$\boldsymbol{param\_value}: \mathbb{F}\ PARAM \rightarrow PNAME \nrightarrow PVALUE$$

$$\forall\ pars: \mathbb{F}\ PARAM;\ pname: PNAME;\ pvalue: PVALUE \bullet$$
$$pname \mapsto pvalue \in param\_value\ pars$$
$$\Leftrightarrow$$
$$(\exists_1\ pv{:}PVALUE \bullet (name \mathrel{\widehat{=}} pname,\ value \mathrel{\widehat{=}} pv) \in pars$$
$$\land\ pv = pvalue)$$

and one for obtaining the used maskvars.

z

$$\boldsymbol{get\_used\_maskvars}: A\_BLOCK \rightarrow \mathbb{F}\ PVALUE$$

$$(\forall\ alb: A\_LIB\_BLOCK \bullet$$
$$get\_used\_maskvars\ (ALibBlock\ alb) = alb.block\_info.used\_maskvars)$$
$$\land$$
$$(\forall\ ass: A\_SUBSYS \bullet$$
$$get\_used\_maskvars\ (ASubsys\ ass) = ass.subsys\_info.used\_maskvars)$$

The used maskvars are be the local (i.e. masked) variables which either occur in the parameters to the block or are free (i.e. masked at a higher level but not at the current level or between the current level and the point of occurrence) in the body of the block (or in the body of any subsystem). The used maskvars fields are set up at the same time as block translation takes place, i.e. during the library look-up phase for library blocks (including block references, in which case the information about maksed variables in the body comes from the metadata), during the subsystem translation phase for subsystems. At present synthesized blocks never use any mask variables.

Initially library blocks are transcribed just copying the parameters, initialising the port detail and leaving all other fields empty.

The port details are initialised from the line information, at this stage all ports to which connections are made are shown as having port type “UnknownPT”. The input port details contain the line name from the line connected to it, if there is one.

The following function is required and informally specified. It takes a PVALUE which is a numeric input port name (i.e. a port number) and converts it to a default signal name of the form “signaln” where n is the port number.

z

$$\boldsymbol{port2signal}: PVALUE \rightarrow PVALUE$$

$$\forall pv: PVALUE \bullet$$
$$port2signal\ pv = sc2pv\ (\texttt{"}signal\texttt{"} \frown pv2sc\ pv)$$

The following function determines a signal name to be used in the port details of an input port (used by bus constructors) from the line name attached to the port. If the line name is empty “signaln” is used, if the line name is enclosed in angled brackets then their content is used, otherwise the line name is used.

z

$$\boldsymbol{force\_signal\_name}: PVALUE \rightarrow PVALUE \rightarrow PVALUE$$

$$\forall\ name, port: PVALUE; name2: \mathbb{F}\ PVALUE$$
$$|\ name2 = \{n: PVALUE \mid pv2sc\ name = \texttt{"<"} \frown (pv2sc\ n) \frown \texttt{">"}\}$$
$$\bullet \quad force\_signal\_name\ port\ name$$
$$= \quad if\ name = NullString$$
$$then\ NullString$$
$$else\ if\ name2 = \{\}\ then\ name\ else\ (\mu n{:}name2)$$

The following specification is inconsistent when its *lines* parameter has two lines connecting to the same destination. Models of this kind are thought never to be created by Simulink and are not supported by ClawZ. An implementation of this specification may use the first it comes across and may give a warning if there is more than one.

z

$$\boldsymbol{initial\_ipds}: PVALUE \to \mathbb{F}\ LINE \to (PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} PORT\_DETAILS)$$

$$\forall\ lines: \mathbb{F}\ LINE;\ blockname: PVALUE$$
$$\bullet\ initial\_ipds\ blockname\ lines =$$
$$\{pv: PVALUE;\ line: lines;\ PORT\_DETAILS$$
$$\mid (block \mathrel{\widehat{=}} blockname,\ port \mathrel{\widehat{=}} pv) \in line.destinations$$
$$\land\ port\_type = UnknownPT$$
$$\land\ line\_name = force\_signal\_name\ pv\ line.name$$
$$\bullet\ pv \mapsto \theta PORT\_DETAILS\}$$

z

$$\boldsymbol{initial\_opds}: PVALUE \to \mathbb{F}\ LINE \to (PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} PORT\_DETAILS)$$

$$\forall\ lines: \mathbb{F}\ LINE;\ blockname: PVALUE$$
$$\bullet\ initial\_opds\ blockname\ lines =$$
$$\{pv: PVALUE;\ line: lines;\ PORT\_DETAILS$$
$$\mid (block \mathrel{\widehat{=}} blockname,\ port \mathrel{\widehat{=}} pv) = line.source$$
$$\land\ port\_type = UnknownPT$$
$$\land\ line\_name = NullString$$
$$\bullet\ pv \mapsto \theta PORT\_DETAILS\}$$

z

$$\boldsymbol{empty\_pi}: PORT\_INFO$$

$$empty\_pi =$$
$$(input\_port\_details \mathrel{\widehat{=}} \{\},$$
$$output\_port\_details \mathrel{\widehat{=}} \{\})$$

z

$$\boldsymbol{initial\_pi}: PVALUE \to \mathbb{F}\ LINE \to PORT\_INFO$$

$$\forall\ lines: \mathbb{F}\ LINE;\ blockname: PVALUE$$
$$\bullet\ initial\_pi\ blockname\ lines =$$
$$(input\_port\_details \mathrel{\widehat{=}} initial\_ipds\ blockname\ lines,$$
$$output\_port\_details \mathrel{\widehat{=}} initial\_opds\ blockname\ lines)$$

z

$$\begin{array}{|l}
\boldsymbol{input\_ports}: (PVALUE \mathrel{\ooalign{$\rightarrow$\cr\hfil$\mid$\hfil}} A\_BLOCK) \rightarrow \mathbb{F}\ PVALUE \\
\hline
\forall\ blockmap: PVALUE \mathrel{\ooalign{$\rightarrow$\cr\hfil$\mid$\hfil}} A\_BLOCK \bullet \\
input\_ports\ blockmap = \\
\\
\{pv: PVALUE;\ alb: A\_LIB\_BLOCK \\
\mid pv \mapsto ALibBlock\ alb \in blockmap \\
\land (sc2pn\ \texttt{"}BlockType\texttt{"}) \mapsto InPort \in (param\_value\ alb.block\_info.pars) \\
\bullet\ pv\}
\end{array}$$

z

$$\begin{array}{|l}
\boldsymbol{libblock\_to\_a\_block}: \mathbb{F}\ PARAM \rightarrow PVALUE \rightarrow \mathbb{F}\ LINE \rightarrow A\_BLOCK \\
\hline
\forall\ pars: \mathbb{F}\ PARAM;\ lines: \mathbb{F}\ LINE;\ pi: PORT\_INFO;\ bi: BLOCK\_INFO;\ bn: PVALUE \\
\mid \quad ((name \mathrel{\widehat{=}} BlockType,\ value \mathrel{\widehat{=}} InPort) \in pars \land pi = initial\_pi\ bn\ lines \\
\quad\quad \lor (name \mathrel{\widehat{=}} BlockType,\ value \mathrel{\widehat{=}} InPort) \notin pars \land pi = empty\_pi) \\
\land \quad bi = (\ pars \mathrel{\widehat{=}} pars, \\
\quad\quad\quad input\_port\_types \mathrel{\widehat{=}} \{\}, \\
\quad\quad\quad output\_port\_types \mathrel{\widehat{=}} \{\}, \\
\quad\quad\quad specification \mathrel{\widehat{=}} \langle\rangle, \\
\quad\quad\quad invocation \mathrel{\widehat{=}} (NoInv,\ (\langle\rangle,\ \langle\rangle,\ \langle\rangle)), \\
\quad\quad\quad virtual \mathrel{\widehat{=}} VUnknown, \\
\quad\quad\quad used\_maskvars \mathrel{\widehat{=}} \{\}) \\
\bullet\ libblock\_to\_a\_block\ pars\ bn\ lines = \\
\quad\quad\quad ALibBlock\ (block\_info \mathrel{\widehat{=}} bi,\ port\_info \mathrel{\widehat{=}} pi)
\end{array}$$

Subsystems are treated similarly, but there are two relevant sets of parameters, those for the subsystem block and those in the enclosed system block. The block map is also set up, containing all the blocks in the subsystem, and the *action_info* is set up, raising as errors any violations of the specified constraints.

We first specify the setup of the action info and the required checks.

The required checks are:

1. Each action subsystem has only one output port.
2. The output of an action subsystem goes only to *Merge* blocks.
3. All inputs to *Merge* blocks come from action subsystems.
4. Action subsystems driven by the same block feed the same *Merge* block.
5. Action subsystems feeding the same *Merge* block are driven by the same block.

6. For the purposes of the above checks a subsystem does not qualify as a block even if the relevant port of the subsystem does connect inside the subsystem to a block of the relevant type.

In this specification the error reporting is informally indicated by supplying a function with the relevant error information. Invocation of this function should result in an error message of the following form:

Action complex checks failed in subsystem *path*

- the following action subsystems do not have exactly one output connection: *err1*
- the following action subsystems have an output connected to something other than a Merge block: *err2*
- the following Merge blocks have an input which does not come from an action subsystem: *err3*
- the following *If* or *SwitchCase* blocks are not each connected through action subsystems to a single Merge block: *err4*
- the following Merge block has inputs from action subsystems which are not all driven by the same *If* or *SwitchCase* block: *err5*

In the above error message any clauses for which the blockname set is empty should be omitted, and if no clauses are included no error report should be made.

z

$$\boldsymbol{action\_subsys\_errors}: (PATH \twoheadrightarrow [err1, err2, err3, err4, err5: \mathbb{F}\ PVALUE])$$

z

$$\boldsymbol{block\_type}: A\_LIB\_BLOCK \rightarrow PVALUE$$

$$\begin{array}{l}
\forall\ alb: A\_LIB\_BLOCK;\ pv: PVALUE \bullet \\
alb \mapsto pv \in block\_type \\
\Leftrightarrow \\
(\exists_1\ pv2: PVALUE \mid (name \mathrel{\widehat{=}} BlockType,\ value \mathrel{\widehat{=}} pv2) \in alb.block\_info.pars \\
\bullet\ pv2 = pv) \\
\lor \neg\ (\exists_1\ pv2: PVALUE \bullet (name \mathrel{\widehat{=}} BlockType,\ value \mathrel{\widehat{=}} pv2) \in alb.block\_info.pars) \\
\quad \land\ pv = NullString
\end{array}$$

In the *make_action_complex* which follows lines to compile a single action complex given the block-name of the root of the complex, the local names should be read:

root	Blockname of an *If* or a *SwitchCase* block
ralb	Root A_LIB_BLOCK
rol	Root Output Lines (lines whose source ports are on the root block)
subsys_map	Map from root portnames to action subsystem blocknames
asol	Action Subsystem Output Lines
merge_map	Map from action subsystem blocknames to Merge block input portnames
asdb	Action Subsystem Destination Blocknames (should be just one Merge block)
tail	Blockname of Merge block

z

$$
\begin{array}{l}
\boldsymbol{make\_action\_complex}: \\
\quad (PVALUE \times (\mathbb{F}\ LINE) \times (PVALUE \mathrel{+\!\!\!\!\!\twoheadrightarrow} A\_BLOCK)) \\
\qquad \mathrel{+\!\!\!\!\rightarrow} ACTION\_COMPLEX \\
\hline
\forall\ root: PVALUE;\ lines: \mathbb{F}\ LINE;\ blockmap: PVALUE \mathrel{+\!\!\!\!\!\twoheadrightarrow} A\_BLOCK; \\
\quad ac: ACTION\_COMPLEX \bullet \\
((root,\ lines,\ blockmap) \mapsto ac) \in make\_action\_complex \\
\Leftrightarrow \\
(\forall\ ralb: A\_LIB\_BLOCK;\ rol,\ asol: \mathbb{F}\ LINE; \\
\quad asdb: \mathbb{F}\ PVALUE;\ type,\ tail: PVALUE;\ open: BOOL; \\
\quad subsys\_map,\ merge\_map: PVALUE \mathrel{+\!\!\!\!\!\twoheadrightarrow} PVALUE \\
\bullet\ root \mapsto (ALibBlock\ ralb) \in blockmap \\
\land\ type = block\_type\ ralb \\
\land\ rol = \{line: lines \mid line.source.block = root\} \\
\land\ subsys\_map = \{sp,\ db: PVALUE;\ dp: PORT;\ line: rol \\
\qquad\qquad \mid sp = line.source.port \\
\qquad\qquad \land\ dp \in line.destinations \\
\qquad\qquad \land\ db = dp.block \\
\qquad\qquad \bullet\ sp \mapsto db\} \\
\land\ asol = \{line: lines \mid line.source.block \in \mathrm{ran}\ subsys\_map\} \\
\land\ merge\_map = \{sb,\ dp1: PVALUE;\ dp2: PORT;\ line: asol \\
\qquad\qquad \mid sb = line.source.block \\
\qquad\qquad \land\ \{dp2\} = line.destinations \\
\qquad\qquad \land\ dp1 = dp2.port \\
\qquad\qquad \bullet\ sb \mapsto dp1\} \\
\land\ asdb = \{port: PORT;\ line: asol \mid port \in line.destinations \bullet port.block\} \\
\land\ \{tail\} = asdb \\
\land\ open = \\
\qquad if\ type = If \\
\qquad then\ (name \mathrel{\widehat=} ShowElse,\ value \mathrel{\widehat=} on) \notin ralb.block\_info.pars \\
\qquad else \\
\qquad\quad if\ type = SwitchCase \\
\qquad\quad then\ (name \mathrel{\widehat=} CaseShowDefault,\ value \mathrel{\widehat=} on) \notin ralb.block\_info.pars \\
\qquad\quad else\ true \\
\land\ ac = (root \mathrel{\widehat=} root,\ type \mathrel{\widehat=} type,\ tail \mathrel{\widehat=} tail,\ open \mathrel{\widehat=} open, \\
\qquad subsys\_map \mathrel{\widehat=} subsys\_map,\ merge\_map \mathrel{\widehat=} merge\_map))
\end{array}
$$

This function specifies how the first five error checks are performed.

Key to names:

roots	Names of *If* or *SwitchCase* blocks
ass	Action subsystem block names
ms	*Merge* block names
err$n$	set of blocks violating check n (as specified above)

Note that a block is deemed an action subsystem for the purposes of these checks if there is a line connecting to an action port on the block. This replaces the previous test for it being a subsystem containing an action port, since a block reference to an action subsystem must also be accepted.

z

$$
\begin{array}{l}
\textbf{action\_complex\_checks}: \\
((\mathbb{F}\ LINE) \times (PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} A\_BLOCK) \times (\mathbb{F}\ PVALUE)) \rightarrow \\
(\mathbb{F}\ PVALUE) \times (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ PVALUE) \\
\hline
\forall\ lines: \mathbb{F}\ LINE; blockmap: PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} A\_BLOCK; roots: \mathbb{F}\ PVALUE; \\
\quad ass, ms, mis: \mathbb{F}\ PVALUE; err1, err2, err3, err4, err5: \mathbb{F}\ PVALUE \\
\mid ass = \quad \{pv: PVALUE; line: lines; port: line.destinations \mid port.port = ActionQ \\
\qquad \land port.block = pv \bullet pv\} \\
\land ms = \quad \{pv: PVALUE; alb: A\_LIB\_BLOCK \mid pv \mapsto ALibBlock\ alb \in blockmap \\
\qquad \land block\_type\ alb = Merge \bullet pv\} \\
\land err1 = \quad \{pv: ass; as: A\_SUBSYS \mid pv \mapsto ASubsys\ as \in blockmap \\
\qquad \land \neg\ (\exists op: PVALUE \bullet\ dom\ as.port\_info.output\_port\_details = \{op\}) \\
\qquad \bullet\ pv\} \\
\land err2 = \quad \{pv: ass; line: lines; d: PORT \\
\qquad \mid line.source.block = pv \\
\qquad \land d \in line.destinations \land d.block \notin ms \bullet pv\} \\
\land err3 = \quad \{line: lines; port: line.destinations \\
\qquad \mid port.block \in ms \land line.source.block \notin ass \bullet port.block\} \\
\land err4 = \quad \{pvr: roots; mbs: \mathbb{F}\ PVALUE \\
\qquad \mid mbs = \quad \{line1, line2: lines; port1, port2: PORT \\
\qquad\qquad \mid line1.source.block = pvr \\
\qquad\qquad \land port1 \in line1.destinations \\
\qquad\qquad \land line2.source.block = port1.block \\
\qquad\qquad \land port2 \in line2.destinations \bullet port2.block\} \\
\qquad \land \neg(\exists\ pvm: ms \bullet \{pvm\} = mbs) \bullet pvr\} \\
\land err5 = \quad \{pvm: ms; sbs: \mathbb{F}\ PVALUE \\
\qquad \mid sbs = \quad \{line1, line2: lines; port1, port2: PORT \\
\qquad\qquad \mid port1 \in line1.destinations \\
\qquad\qquad \land port1.port = ifaction \\
\qquad\qquad \land line2.source.block = port1.block \\
\qquad\qquad \land port2 \in line2.destinations \\
\qquad\qquad \land port2.block = pvm \\
\qquad\qquad \bullet\ line1.source.block\} \\
\qquad \land \neg(\exists\ pvr: roots \bullet \{pvr\} = sbs) \bullet pvm\} \\
\bullet\ action\_complex\_checks\ (lines, blockmap, roots) = (err1, err2, err3, err4, err5)
\end{array}
$$

Key to names:

ifs	*If* block names
ifswe	Names of *If* blocks with *ShowElse* = *on*
scs	*SwitchCase* block names
scswd	Names of *SwitchCase* blocks with *CaseShowDefault* = *on*
acs	map from root blockname to *ACTION_COMPLEX*
err*n*	set of blocks violating check *n* (as specified above)

z

$$
\begin{array}{l}
\boldsymbol{make\_action\_complexes}: \\
\quad ((\mathbb{F}\ LINE) \times (PVALUE \twoheadrightarrow A\_BLOCK) \times PATH) \\
\qquad \rightarrow \mathbb{F}\ ACTION\_COMPLEX \\
\hline
\forall\ lines: \mathbb{F}\ LINE;\ blockmap: PVALUE \twoheadrightarrow A\_BLOCK;\ path: PATH; \\
\ \ ifs, scs, ass, ms: \mathbb{F}\ PVALUE; \\
\ \ err1, err2, err3, err4, err5: \mathbb{F}\ PVALUE; \\
\ \ acs: \mathbb{F}\ ACTION\_COMPLEX \\
\mid ifs = \quad \{pv: PVALUE;\ alb: A\_LIB\_BLOCK \\
\qquad\qquad \mid pv \mapsto ALibBlock\ alb \in blockmap \\
\qquad\qquad \land block\_type\ alb = If \bullet pv\} \\
\land\ scs = \quad \{pv: PVALUE;\ alb: A\_LIB\_BLOCK \\
\qquad\qquad \mid pv \mapsto ALibBlock\ alb \in blockmap \\
\qquad\qquad \land block\_type\ alb = SwitchCase \bullet pv\} \\
\land\ acs = \quad \{pv: ifs \cup scs;\ ac: ACTION\_COMPLEX \\
\qquad\qquad \mid (pv, lines, blockmap) \mapsto ac \in make\_action\_complex \bullet ac\} \\
\land\ (err1, err2, err3, err4, err5) \\
\qquad = action\_complex\_checks\ (lines, blockmap, ifs \cup scs) \\
\land\ path \mapsto (err1 \mathrel{\widehat=} err1, err2 \mathrel{\widehat=} err2, err3 \mathrel{\widehat=} err3, \\
\qquad err4 \mathrel{\widehat=} err4, err5 \mathrel{\widehat=} err5) \\
\qquad \in action\_subsys\_errors \\
\bullet\ make\_action\_complexes\ (lines, blockmap, path) = acs
\end{array}
$$

z

$$
\begin{array}{l}
\textbf{subsys\_template}: HOLD\_CONTEXT \rightarrow PATH \rightarrow \mathbb{F}\ PARAM \rightarrow \mathbb{F}\ PARAM \\
\quad \rightarrow \mathbb{F}\ LINE \rightarrow PORT\_INFO \rightarrow (PVALUE \twoheadrightarrow\!\!\!\!\!\!+\ A\_BLOCK) \rightarrow A\_BLOCK \\
\hline
\forall\ hc: HOLD\_CONTEXT;\ path: PATH;\ pars2,\ pars: \mathbb{F}\ PARAM; \\
ipi: PORT\_INFO;\ blockmap: PVALUE \twoheadrightarrow\!\!\!\!\!\!+\ A\_BLOCK; \\
lines: \mathbb{F}\ LINE;\ acs: \mathbb{F}\ ACTION\_COMPLEX;\ action\_subsys: BOOL \\
\mid acs = make\_action\_complexes\ (lines,\ blockmap,\ path) \\
\land action\_subsys = (\\
\qquad\qquad \{pv: PVALUE;\ alb: A\_LIB\_BLOCK \\
\qquad\qquad \mid pv \mapsto ALibBlock\ alb \in blockmap \\
\qquad\qquad \land block\_type\ alb = ActionPort \\
\qquad\qquad \bullet pv\} \neq \{\}) \\
\bullet subsys\_template\ hc\ path\ pars2\ pars\ lines\ ipi\ blockmap = ASubsys\ (\\
\qquad subsys\_info \mathrel{\widehat{=}} (\\
\qquad\qquad subpars \mathrel{\widehat{=}} pars, \\
\qquad\qquad syspars \mathrel{\widehat{=}} pars2, \\
\qquad\qquad lines \mathrel{\widehat{=}} lines, \\
\qquad\qquad specification \mathrel{\widehat{=}} \langle\rangle, \\
\qquad\qquad invocation \mathrel{\widehat{=}} (NoInv,\ (\langle\rangle,\ \langle\rangle,\ \langle\rangle)), \\
\qquad\qquad virtual \mathrel{\widehat{=}} VUnknown, \\
\qquad\qquad used\_maskvars \mathrel{\widehat{=}} \{\}, \\
\qquad\qquad mv\_ctxt \mathrel{\widehat{=}} \{\}, \\
\qquad\qquad action\_info \mathrel{\widehat{=}} (action\_subsys \mathrel{\widehat{=}} action\_subsys, \\
\qquad\qquad\qquad complexes \mathrel{\widehat{=}} acs,\ held\_context \mathrel{\widehat{=}} hc)), \\
\qquad port\_info \mathrel{\widehat{=}} ipi, \\
\qquad blocks \mathrel{\widehat{=}} blockmap)
\end{array}
$$

A special case in construction of the blockmap is that for a stateflow subsystem. In this case all blocks except port blocks are omitted. The following function discards the unwanted blocks.

z

$$
\begin{array}{l}
\textbf{ports\_only\_blockmap}: (PVALUE \twoheadrightarrow\!\!\!\!\!\!+\ A\_BLOCK) \rightarrow (PVALUE \twoheadrightarrow\!\!\!\!\!\!+\ A\_BLOCK) \\
\hline
\forall\ blockmap: PVALUE \twoheadrightarrow\!\!\!\!\!\!+\ A\_BLOCK \bullet \\
ports\_only\_blockmap\ blockmap = \\
\\
\{pv: PVALUE;\ alb: A\_LIB\_BLOCK;\ bt: port\_block\_types \\
\mid pv \mapsto ALibBlock\ alb \in blockmap \\
\land (sc2pn\ \texttt{"}BlockType\texttt{"}) \mapsto bt \in (param\_value\ alb.block\_info.pars) \\
\bullet pv \mapsto blockmap\ pv\}
\end{array}
$$

The following function returns the set of masked variable for some block, given the block parameters. If the block isn't masked the empty set is returned.

z

$$\mathbf{get\_maskvars}: \mathbb{F}\ PARAM \to \mathbb{F}\ PVALUE$$

$$\begin{array}{l}
\forall\ pars: \mathbb{F}\ PARAM \bullet \\
get\_maskvars\ pars = ran \\
(if\ (sc2pn\ \texttt{"}MaskVariables\texttt{"}) \in dom\ (param\_value\ pars) \\
then\ parse\_maskvar\_param\ (param\_value\ pars\ (sc2pn\ \texttt{"}MaskVariables\texttt{"})) \\
else\ \{\})
\end{array}$$

For support of action subsystems it is necessary to track the *held context* as we recurse through the subsystems in a model. This held context is changed whenever an action subsystem is encountered, according to the *InitialiseStates* parameter on the action block in the subsystem. The following function checks whether the current subsystem contains an action block and updates the held context accordingly.

z

$$\begin{array}{l}
\mathbf{new\_held\_context}: BOOL \times HOLD\_CONTEXT \times (\mathbb{F}\ BLOCK) \\
\qquad \to HOLD\_CONTEXT
\end{array}$$

$$\begin{array}{l}
\forall\ lib: BOOL;\ hc: HOLD\_CONTEXT;\ blocks: \mathbb{F}\ BLOCK; \\
\quad init\_states: \mathbb{F}\ PVALUE \\
\mid init\_states = \\
\qquad \{pv: PVALUE;\ pars: \mathbb{F}\ PARAM \\
\qquad \mid (name \mathrel{\widehat{=}} BlockType,\ value \mathrel{\widehat{=}} ActionPort) \in pars \\
\qquad \land (name \mathrel{\widehat{=}} InitializeStates,\ value \mathrel{\widehat{=}} pv) \in pars \\
\qquad \land LibBlock\ pars \in blocks \\
\qquad \bullet\ pv\} \\
\bullet\ new\_held\_context\ (lib,\ hc,\ blocks) = \\
\qquad if\ init\_states = \{reset\}\ then\ HCReset \\
\qquad else\ if\ init\_states = \{held\}\ then\ HCHeld \\
\qquad else\ if\ init\_states = \{\}\ then \\
\qquad\qquad if\ lib\ then\ HCUnknown\ else\ hc \\
\qquad else\ HCUnknown
\end{array}$$

The following two functions transform a *SYSTEM* into an *A_BLOCK*. A special case is made of systems which have no lines connecting to them. This is intended primarily to cover the input ports at the top level of a model, which are all assumed to be scalars. It also applies to subsystems with no lines attached to input ports, in which case the port details are set up by looking at the port blocks and assuming that they are scalar. This is necessary to give sufficient signal information for signal analysis and block synthesis.

z

$$
\begin{array}{l}
\quad \boldsymbol{blocks\_to\_blockmap}: BOOL \rightarrow HOLD\_CONTEXT \rightarrow PATH \rightarrow \mathbb{F}\ BLOCK \\
\qquad\qquad \rightarrow \mathbb{F}\ LINE \rightarrow (PVALUE \twoheadrightarrow\!\!\!\!\!\shortmid\ \ A\_BLOCK); \\
\quad \boldsymbol{subsystem\_to\_a\_block}: BOOL \rightarrow HOLD\_CONTEXT \rightarrow PATH \rightarrow \mathbb{F}\ PARAM \\
\qquad\qquad \rightarrow PORT\_INFO \rightarrow SYSTEM \rightarrow A\_BLOCK \\
\hline
\forall\ path: PATH \bullet \\
\quad (\forall\ lib: BOOL; hc: HOLD\_CONTEXT; blocks: \mathbb{F}\ BLOCK; lines: \mathbb{F}\ LINE \bullet \\
\quad blocks\_to\_blockmap\ lib\ hc\ path\ blocks\ lines = \\
\quad \{\ pv{:}PVALUE; ab{:}A\_BLOCK; path2: PATH; b{:}blocks; \\
\qquad pi: PORT\_INFO; pars2: \mathbb{F}\ PARAM \\
\quad \mid path2 = path \frown \langle pv \rangle \\
\quad \land\ pi = initial\_pi\ pv\ lines \\
\quad \land\ (\quad (b = LibBlock\ pars2 \\
\qquad \land\ Name \mapsto pv \in param\_value\ pars2 \\
\qquad \land\ ab = libblock\_to\_a\_block\ pars2\ pv\ lines) \\
\quad\ \ \lor \quad (\exists\ sys: SYSTEM \\
\qquad \bullet\ b = SubSystem\ (pars \mathrel{\widehat{=}} pars2, system \mathrel{\widehat{=}} sys) \\
\qquad \land\ Name \mapsto pv \in param\_value\ pars2 \\
\qquad \land\ ab = subsystem\_to\_a\_block\ lib\ hc\ path2\ pars2\ pi\ sys)) \\
\quad \bullet\ (pv \mapsto ab) \\
\quad \}) \\
\\
\land \quad (\forall hc, hc': HOLD\_CONTEXT; pars: \mathbb{F}\ PARAM; system: SYSTEM; \\
\quad\ \ pars2: \mathbb{F}\ PARAM; lines, lines': \mathbb{F}\ LINE; blocks: \mathbb{F}\ BLOCK; \\
\quad\ \ blockmap, blockmap': PVALUE \twoheadrightarrow\!\!\!\!\!\shortmid\ \ A\_BLOCK; lib: BOOL; \\
\quad\ \ ipds': PVALUE \twoheadrightarrow\!\!\!\!\!\shortmid\ \ PORT\_DETAILS; pi: PORT\_INFO \\
\quad \mid system = System\ (pars \mathrel{\widehat{=}} pars2, blocks \mathrel{\widehat{=}} blocks, lines \mathrel{\widehat{=}} lines) \\
\quad \land\ hc' = new\_held\_context\ (lib, hc, blocks) \\
\quad \land\ blockmap = blocks\_to\_blockmap\ lib\ hc'\ path\ blocks\ lines \\
\quad \land\ (blockmap', lines') = \\
\qquad if\ param\_value\ pars\ (sc2pn\ \texttt{"}MaskType\texttt{"}) = sc2pv\texttt{"}Stateflow\texttt{"} \\
\qquad then\ (ports\_only\_blockmap\ blockmap, \{\}) \\
\qquad else\ (blockmap, lines) \\
\quad \bullet\ subsystem\_to\_a\_block\ lib\ hc\ path\ pars\ pi\ system \\
\quad = subsys\_template\ hc'\ path\ pars2\ pars\ lines'\ pi\ blockmap')
\end{array}
$$

Finally we package it up for the top level:

z

$$
\begin{array}{l}
\boldsymbol{system\_to\_a\_block}: BOOL \rightarrow PVALUE \rightarrow SYSTEM \\
\qquad\qquad \rightarrow A\_BLOCK \\
\hline
\forall\ lib: BOOL;\ pv{:}PVALUE;\ sys: SYSTEM;\ pi: PORT\_INFO \\
\mid pi = (input\_port\_details \mathrel{\widehat{=}} \{\}, \\
\qquad output\_port\_details \mathrel{\widehat{=}} \{\}) \\
\bullet\ system\_to\_a\_block\ lib\ pv\ sys \\
\quad = subsystem\_to\_a\_block\ lib\ (if\ lib\ then\ HCUnknown\ else\ HCVoid)\ \langle pv \rangle\ \{\}\ pi\ sys
\end{array}
$$

6.2 Block Traversal Functions

For general use we first define various mapping functions for *A_BLOCK*s.

This version allows accumulation of context and result as well as modification of the *A_BLOCK*. Its argument is a triple of functions which are:

libmap a function for processing a library block

ssmap a function for processing subsystems give the results of processing the blocks in the subsystem

newc a function which computes the new context in which blocks in a subsystem are to be processed

z

$[C, R]$

$$\begin{array}{l}
\boldsymbol{a\_block\_map\_cr}: (C \to A\_LIB\_BLOCK \to R \times A\_LIB\_BLOCK) \\
\quad \times (C \to A\_SUBSYS \times (PVALUE \nrightarrow R) \to R \times A\_SUBSYS) \\
\quad \times (C \to A\_SUBSYS \to PVALUE \to C) \\
\quad \to (C \to A\_BLOCK \nrightarrow R \times A\_BLOCK)
\end{array}$$

$$\begin{array}{l}
\forall\ libmap: C \to A\_LIB\_BLOCK \to R \times A\_LIB\_BLOCK; \\
\quad ssmap: C \to A\_SUBSYS \times (PVALUE \nrightarrow R) \to R \times A\_SUBSYS; \\
\quad newc: C \to A\_SUBSYS \to PVALUE \to C; \\
\quad c: C \bullet \\
(\forall\ alb, alb': A\_LIB\_BLOCK;\ r{:}R \\
\mid (r, alb') = libmap\ c\ alb \\
\bullet\ a\_block\_map\_cr\ (libmap,ssmap,newc)\ c\ (ALibBlock\ alb) = (r, ALibBlock\ alb')) \\
\land \\
(\forall\ ass, ass', ass'': A\_SUBSYS;\ r{:}R;\ rabm: PVALUE \nrightarrow (R \times A\_BLOCK); \\
\quad rm: PVALUE \nrightarrow R;\ abm: PVALUE \twoheadrightarrow A\_BLOCK \\
\mid rabm = \quad \{pv: PVALUE;\ r: R;\ ab, ab': A\_BLOCK;\ c': C \\
\qquad \mid pv \mapsto ab \in ass.blocks \land c' = newc\ c\ ass\ pv \\
\qquad \land a\_block\_map\_cr\ (libmap,ssmap,newc)\ c'\ ab = (r, ab') \\
\qquad \bullet\ pv \mapsto (r, ab')\} \\
\land rm = rabm \fatsemi first \\
\land abm = rabm \fatsemi second \\
\land ass' = (\quad subsys\_info \mathrel{\widehat=} ass.subsys\_info, \\
\qquad port\_info \mathrel{\widehat=} ass.port\_info, \\
\qquad blocks \mathrel{\widehat=} abm) \\
\land (r, ass'') = ssmap\ c\ (ass', rm) \\
\bullet\ a\_block\_map\_cr\ (libmap,ssmap,newc)\ c\ (ASubsys\ ass) = (r, ASubsys\ ass''))
\end{array}$$

This one updates the *A_BLOCK* without returning a result.

z

$$\begin{array}{l}
\boldsymbol{a\_block\_map}: (PATH \to A\_BLOCK \pfun A\_BLOCK) \\
\quad \to PATH \to A\_BLOCK \pfun A\_BLOCK \\
\hline
\forall m: PATH \to A\_BLOCK \pfun A\_BLOCK;\ path: PATH;\ ab, ab': A\_BLOCK \bullet \\
\quad ab \mapsto ab' \in (a\_block\_map\ m\ path) \\
\Leftrightarrow \\
\quad (\exists alb: A\_LIB\_BLOCK \mid ab = ALibBlock\ alb \bullet ab \mapsto ab' \in (m\ path)) \\
\lor \\
\quad (\exists as, as': A\_SUBSYS;\ blocks, blocks': PVALUE \ffun A\_BLOCK \\
\quad \mid ab = ASubsys\ as \\
\quad \land blocks = as.blocks \\
\quad \land blocks' = \quad \{bname: PVALUE;\ ablock, ablock': A\_BLOCK \\
\quad\quad\quad \mid bname \mapsto ablock \in blocks \\
\quad\quad\quad \land ablock \mapsto ablock' \in (a\_block\_map\ m\ (path \frown \langle bname \rangle)) \\
\quad\quad\quad \bullet bname \mapsto ablock'\} \\
\quad \land as' = \quad (subsys\_info \widehat{=} as.subsys\_info, \\
\quad\quad\quad port\_info \widehat{=} as.port\_info, \\
\quad\quad\quad blocks \widehat{=} blocks \oplus blocks') \\
\quad \bullet ASubsys\ as' \mapsto ab' \in m\ path)
\end{array}$$

The following function selects a block using a path.

z

$$\begin{array}{l}
\boldsymbol{a\_block\_select}: PATH \to A\_BLOCK \pfun A\_BLOCK \\
\hline
\quad \forall p{:}PATH;\ ab{,}ab': A\_BLOCK \bullet \\
\quad (ab, ab') \in a\_block\_select\ p \\
\quad \Leftrightarrow \\
\quad p = \langle\rangle \land ab' = ab \\
\quad \lor \\
\quad (\exists ass: A\_SUBSYS;\ bs: PVALUE \ffun A\_BLOCK;\ ab'': A\_BLOCK \bullet \\
\quad ab = ASubsys\ ass \land bs = ass.blocks \land (head\ p \mapsto ab'') \in bs \\
\quad \land ab'' \mapsto ab' \in a\_block\_select\ (tail\ p))
\end{array}$$

6.3 Ports, Lines and Equations

When considering a Simulink system it is necessary to distinguish between ports internal to the diagram defining the system, which are not accessible at the higher level when the system is used as a subsystem, and ports which are external to the system and will appear as ports on the block used to incorporate the system as a subsystem of a larger system.

When a system diagram is viewed, the lines on the diagram connect internal ports. External ports are blocks in the diagram (from the connections library), each having a single internal port. To make a connection to an external port, a connection is made to the internal port on the appropriate external port library block.

Though external port blocks in a Simulink subsystem have names, these names are not used when connection are made to the port in the system containing the subsystem. Furthermore, no names are available for instances of library blocks. For our purposes ports in Simulink are therefore numbered rather than named.

The port configuration of a Simulink System is shown by the following information in the model:

1. A *Ports* parameter for each block consisting of a 5-tuple of numeric port counts, giving the highest numbered input and output ports and 0 or 1 to indicate the absence or presence respectively of enable, trigger and state ports. Though this usually gives the *numbers* of input and output ports, this is not always the case. There may be gaps in the port numbers. At most one each of the latter three are permitted. Trailing zeros are now usually omitted. The information is represented abstractly in this model by the structure *PORT_PARAM*
2. Input and Output Port blocks from the connection library. These enable the external ports of a system to be shown as blocks with a single internal port each, enabling the connections to external ports to be shown by connection to these internal ports.
3. Trigger and Enable blocks

   These blocks, which have blocktype “TriggerPort” and “EnablePort” respectively, are included in a subsystem to make that subsystem triggered or enabled respectively. Unlike input and output ports no connection can be made to these ports from within the subsystem.
4. The line information showing how the external ports are connected internally.

6.3.1 Port Naming Conventions

Note that in the Simulink model the only way to tell whether a line is connecting to an input port or an output port (on a library or subsystem block) is by observing which side of the line the port is, i.e. lines always connect their source to an output port and their destinations to input ports. On the Simulink diagram the output port is shown by a blob and the input port by an arrowhead on the line.

This uniform naming convention will generate a lot of potential name clashes in the Z, since all the blocks in the diagram will re-use the same names for their inputs and outputs. It is proposed that a renaming algorithm based on adding a prefix be used. In the first instance the prefix will be the block name.

The following functions map port numbers (as PVALUEs) to the corresponding IDENTs, with special cases for non-numeric input ports.

Note that numbered action ports do not appear in port details and are therefore not covered by the following specification.

z

$$
\begin{array}{l}
\boldsymbol{inport\_name}: PVALUE \rightarrow IDENT \\
\hline
\forall\ port: PVALUE \bullet \\
inport\_name\ port = \\
(\mu i: IDENT;\ n: \mathbb{N};\ s: seq\ CHAR \\
\mid port = sc2pv\ \texttt{"}trigger\texttt{"} \land i = pvalue2ident\ TriggerQ \\
\lor port = sc2pv\ \texttt{"}enable\texttt{"} \land i = pvalue2ident\ EnableQ \\
\lor port = ifaction \land i = pvalue2ident\ ActionQ \\
\lor port = num2pvalue\ n \\
\quad \land sc2pv\ s = num2pvalue\ n \\
\quad \land i = pvalue2ident\ (sc2pv\ (\texttt{"}In\texttt{"} \frown s \frown \texttt{"?"})) \\
\bullet\ i\ )
\end{array}
$$

z

$$
\begin{array}{l}
\boldsymbol{outport\_name}: PVALUE \rightarrow IDENT \\
\hline
\forall\ port: PVALUE \bullet \\
outport\_name\ port = \\
(\mu\ oi: IDENT;\ n: \mathbb{N};\ s: seq\ CHAR \\
\mid port = num2pvalue\ n \\
\quad \land sc2pv\ s = num2pvalue\ n \\
\quad \land oi = pvalue2ident\ (sc2pv\ (\texttt{"}Out\texttt{"} \frown s \frown \texttt{"!"})) \\
\bullet\ oi\ )
\end{array}
$$

6.3.2 Equations

The set of equations is primarily derived from the set of *LINE*s in the diagram, but with virtualization and support for action subsystems other sources of information are also necessary. These include the information about which blocks have been virtualized and the results of the virtualization (which may be thought of as functionality absorbed into the wiring equations), and information about the action subsystem complexes (which is used to create additional connections transferring the activation state of action ports to the relevant merge blocks).

The model coordinate positions (block/port) have to be translated into a compound name consisting of the block name (local to and as specified on the diagram, subject to sanitization) followed by a dot followed by the relevant conventional port component name. Where the block is an external port a simple portname is required. To tell which case applies it is necessary to know the type of each block, this information is supplied by a parameter of type *BLOCK_TYPES*.

Each line has its ports specified as block/port combinations.

The following function expresses the claim that a *PORT* p is in fact a connection to an input port block whose *IDENT* should be i, in the context of a given set of blocks. It also covers *TriggerPort*, *EnablePort* and *ActionPort*.

z

$$
\begin{array}{l}
\boldsymbol{xinport\_ident}: (\mathbb{F}\ A\_BLOCK) \rightarrow \mathbb{P}\ (PORT \times IDENT) \\
\hline
\forall\ blocks: \mathbb{F}\ A\_BLOCK;\ p: PORT;\ i: IDENT\ \bullet \\
(p,i) \in (xinport\_ident\ \ blocks) \\
\Leftrightarrow \\
(\exists\ bi: BLOCK\_INFO;\ pi: PORT\_INFO;\ params: \mathbb{F}\ PARAM;\ pp: PVALUE \\
\bullet \quad ALibBlock\ (block\_info \mathrel{\widehat{=}} bi,\ port\_info \mathrel{\widehat{=}} pi) \in blocks \\
\wedge \quad params = bi.pars \\
\wedge \quad (name \mathrel{\widehat{=}} Name,\ value \mathrel{\widehat{=}} p.block) \in params \\
\wedge \quad (\quad (name \mathrel{\widehat{=}} BlockType,\ value \mathrel{\widehat{=}} InPort) \in params \\
\qquad\qquad \wedge\ (name \mathrel{\widehat{=}} Port,\ value \mathrel{\widehat{=}} pp) \in params \\
\qquad\qquad \wedge\ i = inport\_name\ (strip\_pval\ \ pp) \\
\qquad \vee \quad (name \mathrel{\widehat{=}} BlockType,\ value \mathrel{\widehat{=}} TriggerPort) \in params \\
\qquad\qquad \wedge\ i = pvalue2ident\ \ TriggerQ \\
\qquad \vee \quad (name \mathrel{\widehat{=}} BlockType,\ value \mathrel{\widehat{=}} EnablePort) \in params \\
\qquad\qquad \wedge\ i = pvalue2ident\ \ EnableQ \\
\qquad \vee \quad (name \mathrel{\widehat{=}} BlockType,\ value \mathrel{\widehat{=}} ActionPort) \in params \\
\qquad\qquad \wedge\ i = pvalue2ident\ \ ActionQ \\
\qquad) \\
)
\end{array}
$$

The following function expresses the claim that a *PORT* p is in fact a connection to an output port block whose *IDENT* should be i, in the context of a give set of blocks.

z

$$
\begin{array}{l}
\boldsymbol{xoutport\_ident}: (\mathbb{F}\ A\_BLOCK) \rightarrow \mathbb{P}\ (PORT \times IDENT) \\
\hline
\forall\ blocks: \mathbb{F}\ A\_BLOCK;\ p: PORT;\ i: IDENT \bullet \\
(p,i) \in (xoutport\_ident\ blocks) \\
\Leftrightarrow \\
(\exists\ bi: BLOCK\_INFO;\ pi: PORT\_INFO;\ params: \mathbb{F}\ PARAM;\ pp: PVALUE \\
\bullet \quad ALibBlock\ (block\_info \mathrel{\widehat{=}} bi,\ port\_info \mathrel{\widehat{=}} pi) \in blocks \\
\land \quad params = bi.pars \\
\land \quad (name \mathrel{\widehat{=}} BlockType,\ value \mathrel{\widehat{=}} OutPort) \in params \\
\land \quad (name \mathrel{\widehat{=}} Name,\ value \mathrel{\widehat{=}} p.block) \in params \\
\land \quad (name \mathrel{\widehat{=}} Port,\ value \mathrel{\widehat{=}} pp) \in params \\
\land \quad i = outport\_name\ (strip\_pval\ pp) \\
)
\end{array}
$$

The identifier used for non-port blocks is retrieved from the *INVKEY* in the *INVOCATION* of the block. Previously this was derived from the path to the block, but this fails when referring to a block in the main system from an artificial subsystem which does not change that block (because the identifier is resolved in the wrong context). The following function specifies how the required identifier is retrieved:

z

$$
\begin{array}{l}
\boldsymbol{get\_invocation}: A\_BLOCK \rightarrow INVOCATION \\
\hline
(\forall\ alb: A\_LIB\_BLOCK \bullet \\
get\_invocation\ (ALibBlock\ alb) = alb.block\_info.invocation) \\
\land \\
(\forall\ ass: A\_SUBSYS \bullet \\
get\_invocation\ (ASubsys\ ass) = ass.subsys\_info.invocation)
\end{array}
$$

This function obtains the identifier for a (non input port) block from the invocation key (*INVKEY*) of the block.

z

$$
\begin{array}{l}
\boldsymbol{block\_ident}: PATH \rightarrow (PVALUE \mathrel{\rightarrow\!\!\!\!\!\!\mapsto} A\_BLOCK) \rightarrow PVALUE \rightarrow IDENT \\
\hline
\forall\ path: PATH;\ blocks: PVALUE \mathrel{\rightarrow\!\!\!\!\!\!\mapsto} A\_BLOCK;\ blockname: PVALUE \bullet \\
block\_ident\ path\ blocks\ blockname = \\
(\mu\ i: IDENT;\ invkey: INVKEY \\
\mid invkey = (first\ o\ get\_invocation)\ (blocks\ blockname) \\
\land\ (OtherInv\ i = invkey \\
\quad \lor\ (invkey = NoInv \land i = path2loci\ (path \frown \langle blockname \rangle))) \\
\bullet\ i)
\end{array}
$$

Next we define the functions which translate each *PORT* into a *Z_EXPR*.

z

$$
\begin{array}{l}
\boldsymbol{eitherport\_ident}: ((\mathbb{F}\ A\_BLOCK) \rightarrow \mathbb{P}\ (PORT \times IDENT)) \rightarrow \\
\quad (PVALUE \rightarrow IDENT) \rightarrow \\
\quad PATH \rightarrow (PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} A\_BLOCK) \rightarrow PORT \rightarrow Z\_EXPR \\
\hline
\forall\ xotherport\_ident: (\mathbb{F}\ A\_BLOCK) \rightarrow \mathbb{P}\ (PORT \times IDENT); \\
\quad name\_fnc: PVALUE \rightarrow IDENT; \\
\quad path: PATH;\ blocks: PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} A\_BLOCK;\ port: PORT \bullet \\
\qquad (\forall\ i: IDENT \mid (port, i) \in (xotherport\_ident\ (ran\ blocks)) \\
\qquad \bullet\ eitherport\_ident\ xotherport\_ident\ name\_fnc\ path\ blocks\ port = Ident\ i) \\
\land \qquad (\neg\ (\exists\ i: IDENT \bullet (port, i) \in (xotherport\_ident\ (ran\ blocks))) \\
\qquad \Rightarrow \\
\qquad eitherport\_ident\ xotherport\_ident\ name\_fnc\ path\ blocks\ port \\
\qquad = Selection\ (\ Ident\ (block\_ident\ path\ blocks\ port.block), \\
\qquad\qquad\qquad Ident\ (name\_fnc\ port.port)) \\
\qquad)
\end{array}
$$

z

$$
\begin{array}{l}
\boldsymbol{inport\_ident}: PATH \rightarrow (PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} A\_BLOCK) \rightarrow PORT \rightarrow Z\_EXPR; \\
\boldsymbol{outport\_ident}: PATH \rightarrow (PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} A\_BLOCK) \rightarrow PORT \rightarrow Z\_EXPR \\
\hline
inport\_ident = eitherport\_ident\ xinport\_ident\ inport\_name; \\
outport\_ident = eitherport\_ident\ xoutport\_ident\ outport\_name
\end{array}
$$

The outputs from virtualized blocks are expressions specified as follows.

First we define a function to extract the map from the *virtual* field of a block.

z

$$
\begin{array}{l}
\boldsymbol{virtual\_map}: A\_BLOCK \mathrel{+\!\!\!\!\rightarrow} (PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} Z\_EXPR) \\
\hline
\forall\ block: A\_BLOCK;\ map: PVALUE \mathrel{+\!\!\!\!\rightarrow} Z\_EXPR \bullet \\
\quad block \mapsto map \in virtual\_map \\
\quad \Leftrightarrow \\
\quad (\exists\ alb: A\_LIB\_BLOCK \\
\quad \mid ALibBlock\ alb = block \\
\quad \bullet\ alb.block\_info.virtual = Virtual\ map) \\
\quad \lor \\
\quad (\exists\ ass: A\_SUBSYS \\
\quad \mid ASubsys\ ass = block \\
\quad \bullet\ ass.subsys\_info.virtual = Virtual\ map)
\end{array}
$$

Note that the following function is partial.

z

$$\boldsymbol{outport\_expression}: (PVALUE \twoheadrightarrow\!\!\!\!\!\!\mapsto A\_BLOCK) \rightarrow PORT \mapsto\!\!\!\!\!\!\rightarrow Z\_EXPR$$

$$\begin{array}{l}
\forall\ blocks: PVALUE \twoheadrightarrow\!\!\!\!\!\!\mapsto A\_BLOCK;\ port: PORT;\ ze: Z\_EXPR \bullet \\
\quad port \mapsto ze \in outport\_expression\ blocks \\
\quad \Leftrightarrow \\
\quad (\exists\ block: A\_BLOCK;\ map: PVALUE \mapsto\!\!\!\!\!\!\rightarrow Z\_EXPR \\
\quad \mid port.block \mapsto block \in blocks \\
\quad \land\ block \mapsto map \in virtual\_map \\
\quad \bullet\ port.port \mapsto ze \in map)
\end{array}$$

Then we define how an equation can be derived from a LINE.

There are two cases to consider, simple equations between line names corresponding to connections between non-virtual blocks, and equations giving values to non-virtual block input ports as expressions for the values of the relevant output port of a virtual block.

It is useful here to define the set of virtual blocks:

z

$$\boldsymbol{virtual\_block}: \mathbb{P}\ A\_BLOCK$$

$$\begin{array}{l}
virtual\_block = \\
\quad \{b: A\_BLOCK;\ ass: A\_SUBSYS;\ alb: A\_LIB\_BLOCK; \\
\quad\ v: VIRTUAL;\ zemap: PVALUE \twoheadrightarrow\!\!\!\!\!\!\mapsto Z\_EXPR \\
\quad \mid (b = ALibBlock\ alb \land v = alb.block\_info.virtual \\
\quad\quad \lor\ b = ASubsys\ ass \land v = ass.subsys\_info.virtual) \\
\quad \land\ v = Virtual\ zemap \\
\quad \bullet\ b\}
\end{array}$$

The final source of information for line equations is the *ACTION_INFO* which contains information about the action subsystems sufficient to permit extra connections to be made from their action ports to ports on the relevant merge blocks. In fact we connect the merge blocks to the source of the signal at the action ports, which will always be an output port on an *If* or *SwitchCase* block.

The ClawZ implementation of *Merge* blocks includes an extra port for each input port n which is named "Action$n$?", the following function maps an input port name on a merge block to the namer of the corresponding action input port.

z

$$\begin{array}{l}
\boldsymbol{merge\_action\_port\_name}: PVALUE \rightarrow IDENT \\
\hline
\forall p: PVALUE; ident: IDENT; s: seq\ CHAR \bullet \\
\quad p \mapsto ident \in merge\_action\_port\_name \\
\Leftrightarrow \\
\quad p = sc2pv\ s \\
\quad \land\ ident = pvalue2ident\ (sc2pv\ (\texttt{"}Action\texttt{"} \frown s \frown \texttt{"?"}))
\end{array}$$

z

$$\begin{array}{l}
\boldsymbol{merge\_action\_port\_ident}: PATH \rightarrow (PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} A\_BLOCK) \\
\qquad \rightarrow PORT \rightarrow Z\_EXPR \\
\hline
\forall path: PATH; blocks: PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} A\_BLOCK; p: PORT \bullet \\
\quad merge\_action\_port\_ident\ path\ blocks\ p \\
\quad = Selection(\quad Ident(block\_ident\ path\ blocks\ p.block), \\
\qquad\qquad Ident(merge\_action\_port\_name\ p.port))
\end{array}$$

This function is not currently used in this specification.

z

$$\begin{array}{l}
\boldsymbol{get\_port\_info}: A\_BLOCK \rightarrow PORT\_INFO \\
\hline
\quad (\forall alb: A\_LIB\_BLOCK \bullet \\
\quad get\_port\_info\ (ALibBlock\ alb) = alb.port\_info) \\
\quad \land \\
\quad (\forall ass: A\_SUBSYS \bullet \\
\quad get\_port\_info\ (ASubsys\ ass) = ass.port\_info)
\end{array}$$

It is necessary to inhibit the generation of equations which refer to ports which have been filtered out of an artificial subsystem. This is done using port check functions which refer to the *port_info* for the block.

The following functions specify the tests.

z

$$\begin{array}{l}
\boldsymbol{input\_port\_names}: A\_BLOCK \rightarrow \mathbb{P}\ PVALUE \\
\hline
\quad \forall ab: A\_BLOCK; pv: PVALUE \bullet \\
\quad pv \in input\_port\_names\ ab \\
\quad \Leftrightarrow \\
\quad\ \ (\exists alb: A\_LIB\_BLOCK \bullet ab = ALibBlock\ alb) \\
\quad \lor (\exists as: A\_SUBSYS \bullet ab = ASubsys\ as \\
\qquad \land\ pv \in dom\ as.port\_info.input\_port\_details)
\end{array}$$

z

$$\textbf{\textit{output\_port\_names}}: A\_BLOCK \rightarrow \mathbb{P}\ PVALUE$$

$$\begin{array}{l}
\forall\ ab: A\_BLOCK;\ pv: PVALUE \bullet \\
pv \in output\_port\_names\ ab \\
\Leftrightarrow \\
\quad (\exists\ alb: A\_LIB\_BLOCK \bullet\ ab = ALibBlock\ alb) \\
\lor\ (\exists\ as: A\_SUBSYS \bullet\ ab = ASubsys\ as \\
\qquad \land\ pv \in dom\ as.port\_info.output\_port\_details)
\end{array}$$

z

$$\textbf{\textit{valid\_source\_ports}}: (PVALUE \twoheadrightarrow\!\!\!\!\!\!\mapsto A\_BLOCK) \rightarrow \mathbb{F}\ PORT$$

$$\begin{array}{l}
\forall\ blocks: PVALUE \twoheadrightarrow\!\!\!\!\!\!\mapsto A\_BLOCK \bullet \\
valid\_source\_ports\ blocks = \\
\qquad \{block: dom\ blocks;\ ab: A\_BLOCK;\ port: PVALUE \\
\qquad \mid ab = blocks\ block \\
\qquad \land\ port \in output\_port\_names\ ab \\
\qquad \bullet\ \theta PORT\}
\end{array}$$

z

$$\textbf{\textit{valid\_destination\_ports}}: (PVALUE \twoheadrightarrow\!\!\!\!\!\!\mapsto A\_BLOCK) \rightarrow \mathbb{F}\ PORT$$

$$\begin{array}{l}
\forall\ blocks: PVALUE \twoheadrightarrow\!\!\!\!\!\!\mapsto A\_BLOCK \bullet \\
valid\_destination\_ports\ blocks = \\
\qquad \{block: dom\ blocks;\ ab: A\_BLOCK;\ port: PVALUE \\
\qquad \mid ab = blocks\ block \\
\qquad \land\ port \in input\_port\_names\ ab \\
\qquad \bullet\ \theta PORT\}
\end{array}$$

The following function determines the set of supplementary destinations for a line which starts at a port on the root of an action complex. This will be either empty or a singleton set which is the relevant action port on the merge block to which the source port is connected through some action subsystem.

z

$$
\begin{array}{l}
\boldsymbol{merge\_dests}: PATH \rightarrow (PVALUE \nrightarrow A\_BLOCK) \rightarrow \\
\qquad (\mathbb{F}\ ACTION\_COMPLEX) \rightarrow PORT \rightarrow (\mathbb{F}\ Z\_EXPR) \\
\hline
\forall\ blocks: PVALUE \nrightarrow A\_BLOCK;\ lines: \mathbb{F}\ LINE; \\
\quad path: PATH;\ acs: \mathbb{F}\ ACTION\_COMPLEX;\ sourceport: PORT \\
\bullet\ merge\_dests\ path\ blocks\ acs\ sourceport = \\
\{\ ac: ACTION\_COMPLEX;\ mergeport: PVALUE \\
\mid ac.root = sourceport.block \\
\land\ sourceport.port \mapsto mergeport \in ac.subsys\_map \operatorname{⨾} ac.merge\_map \\
\bullet\ merge\_action\_port\_ident\ path\ blocks\ (block \mathrel{\widehat{=}} ac.tail,\ port \mathrel{\widehat{=}} mergeport)\}
\end{array}
$$

Now we deal with the non-virtual connections (incorporating action connections).

This is a partial map which delivers a result only if the LINE starts from a present non-virtual block and connects to at least one block which is present and non-virtual. The check is made against a block map which may have been restricted if we are processing an artificial subsystem.

z

$$
\begin{array}{l}
\boldsymbol{line\_equations}: PATH \rightarrow (PVALUE \nrightarrow A\_BLOCK) \\
\qquad \rightarrow (\mathbb{F}\ ACTION\_COMPLEX) \rightarrow LINE \nrightarrow Z\_PRED \\
\hline
\forall\ blocks: PVALUE \nrightarrow A\_BLOCK;\ line: LINE;\ srce: Z\_EXPR; \\
\quad dests: \mathbb{F}_1\ Z\_EXPR;\ path: PATH;\ acs: \mathbb{F}\ ACTION\_COMPLEX\ \bullet \\
\qquad (line \mapsto PredEq(srce,\ dests)) \in line\_equations\ path\ blocks\ acs \\
\Leftrightarrow \\
\qquad line.source \in valid\_source\_ports\ blocks \\
\land \qquad (blocks\ line.source.block) \notin virtual\_block \\
\land \qquad srce = outport\_ident\ path\ blocks\ line.source \\
\land \qquad dests = \qquad \{p: PORT \\
\qquad\qquad\qquad\qquad \mid p \in line.destinations \\
\qquad\qquad\qquad\qquad \land\ p \in valid\_destination\_ports\ blocks \\
\qquad\qquad\qquad\qquad \land\ (blocks\ p.block) \notin virtual\_block \\
\qquad\qquad\qquad\qquad \bullet\ inport\_ident\ path\ blocks\ p\} \\
\qquad\qquad\quad \cup\ (merge\_dests\ path\ blocks\ acs\ line.source) \\
\land \qquad dests \neq \qquad \{\}
\end{array}
$$

Now we specify the equations for non-virtual inputs connected to virtual outputs.

z

$$
\begin{array}{|l}
\quad \boldsymbol{virtual\_line\_equations}: PATH \rightarrow (PVALUE \twoheadrightarrow\!\!\!\!\!\shortmid\; A\_BLOCK) \rightarrow LINE \\
\qquad\qquad \twoheadrightarrow\!\!\!\!\!\shortmid\; Z\_PRED \\
\hline
\forall\ blocks: PVALUE \twoheadrightarrow\!\!\!\!\!\shortmid\; A\_BLOCK;\ line: LINE;\ srce: Z\_EXPR; \\
\quad dests: \mathbb{F}_1\ Z\_EXPR;\ path: PATH \bullet \\
\qquad (line \mapsto PredEq(srce,\ dests)) \in virtual\_line\_equations\ path\ blocks \\
\Leftrightarrow \\
\qquad line.source \in valid\_source\_ports\ blocks \\
\land \qquad (blocks\ line.source.block) \in virtual\_block \\
\land \qquad line.source \mapsto srce \in outport\_expression\ blocks \\
\land \qquad dests = \qquad \{p: PORT \\
\qquad\qquad\qquad\qquad \mid p \in line.destinations \\
\qquad\qquad\qquad\qquad \land\ p \in valid\_destination\_ports\ blocks \\
\qquad\qquad\qquad\qquad \land\ (blocks\ p.block) \notin virtual\_block \\
\qquad\qquad\qquad\qquad \bullet\ inport\_ident\ path\ blocks\ p\} \\
\land \qquad dests \neq \qquad \{\}
\end{array}
$$

Finally we combine the production of virtual equations and line equations into a single specification. An implementation may consider issuing a warning about one-ended lines, though this lies outside the scope of this formal specification. Though not formally specified it is required that the virtual equations appear first in the resulting predicate, then normal line equations, and last action equations.

z

$$
\begin{array}{|l}
\quad \boldsymbol{lines\_equations}: PATH \rightarrow (PVALUE \twoheadrightarrow\!\!\!\!\!\shortmid\; A\_BLOCK) \rightarrow (\mathbb{F}\ LINE) \\
\qquad \rightarrow (\mathbb{F}\ ACTION\_COMPLEX) \rightarrow Z\_PRED \\
\hline
\quad \forall\ blocks: PVALUE \twoheadrightarrow\!\!\!\!\!\shortmid\; A\_BLOCK;\ lines: \mathbb{F}\ LINE; \\
\qquad path: PATH;\ acs: \mathbb{F}\ ACTION\_COMPLEX \\
\quad \bullet\ lines\_equations\ path\ blocks\ lines\ acs = PredConj(\\
\qquad\quad (virtual\_line\_equations\ path\ blocks\ (\!|\ lines\ |\!)) \\
\qquad\quad \cup\ (line\_equations\ path\ blocks\ acs\ (\!|\ lines\ |\!)))
\end{array}
$$

6.4 Library Metadata Matching and Block Instantiation

Block which are not subsystems will be matched against available library specifications. The effects of parameterisation of Simulink library blocks are complex, and early prototypes of the translator will be limited in their ability cope with complex parameterisation. A simple but flexible method is proposed as an interim pending a fuller evaluation of the full range of effects which can be achieved by parameterisation.

The basic idea is to permit any set of parameters to be specified as preconditions for the use of a library definition, without requiring a one-one mapping between Simulink block types and Z definitions. Where Simulink realises a kind of overloading by parameterisation, this can be translated by providing several Z specifications. Control of which specification is selected for any particular instance of the block type can then be achieved by specifying appropriate parameter values in the library metadata. For example, the selection may be based on the number of input and output ports to the instantiated block.

Where complex effects are achieved by parameterisation in Simulink, these affects may be realised in different places in the proposed architecture. An effect may be realised by special provision in the code of the translator, by the provision of extra information in the library metadata or by inference from the form of definitions available in the relevant Z library. It is preferred where possible that no special provision for a Simulink library block be made beyond providing an appropriate Z specification for that block. Where for technical reasons this cannot suffice, it is preferred that supplementary information be injected into the library metadata, rather than additional complexity in the translator code. It is expected however, that to cope with the full range of features in the Simulink library all three methods will be required.

6.4.1 Matching

There are three stages in matching.

In the first stage a potential match will be rejected if the metadata contains a "BlockPath" parameter which does not match the path of the block for which the match is sought.

In the second stage potential matches are selected from the metafile by comparing the set of parameters provided to instantiate the block in the Simulink model with the set of selection parameters specified with each *META_ELEMENT* in the *META_FILE*.

If this selection criterion is passed, the match may later be rejected by the parameter translation facilities.

The function *match* assists in checking whether the meta-elements associated with a library are consistent with the parameters on some block in the Simulink model.

z

$$\boldsymbol{match}: PVALUE \rightarrow \mathbb{F}\ PARAM \rightarrow \mathbb{P}\ META\_ELEMENT$$

$$\begin{array}{l}
\forall\ as\_name: PVALUE;\ pars: \mathbb{F}\ PARAM \bullet \\
match\ as\_name\ pars = \\
\qquad \{me: META\_ELEMENT \\
\qquad \mid me.select\_pars \subseteq pars \\
\qquad \land\ me.as\_name = as\_name\}
\end{array}$$

The function *path_match* checks whether a PATH complies with a PATTERN.

z

$$\begin{array}{l}
\boldsymbol{path\_match}: \mathbb{P}\ (PATH \times PATTERN) \\
\hline
\forall\ p: PATH;\ bp: PATTERN \bullet \\
(p,\ bp) \in path\_match \Leftrightarrow \\
(\exists sspv: seq\ seq\ PVALUE \mid \#sspv = \#bp \\
\bullet\ \frown/\ sspv = p \\
\land\ (\forall n: dom(sspv);\ pv: PVALUE \bullet \\
\qquad\qquad bp\ n = QueryPat \Rightarrow \#(sspv\ n) = 1 \\
\qquad \land \qquad bp\ n = PlainPat\ pv \Rightarrow sspv\ n = \langle pv \rangle) \\
)
\end{array}$$

6.4.2 Library Block Instantiation

This method can either succeed or it can fail in either of two ways. It can fail the match or accept the match and raise errors in instantiation.

The following free type is used for the result returned from the instantiation function.

z

$$\begin{array}{l}
\boldsymbol{INSTANTIATION\_RESULT} ::= \\
\qquad\qquad \boldsymbol{IMatch}\ BLOCK\_INFO \\
\qquad \mid \qquad \boldsymbol{INoMatch} \\
\qquad \mid \qquad \boldsymbol{IFail}
\end{array}$$

The following function will create a *definition* for an instance of a library block and a *declaration* which can be used to invoke that definition. Its primary information source is the matching *META_ELEMENT*, but it also takes the blockname and a context as parameters, since these are required to determine both the name of the schema defined and the name of the variable used when subsequently invoking the definition.

In addition it is necessary to know which of the variables used in parameters transmitted to the block are *local* i.e. are in the scope of a masked subsystem with a variable of that name. If there are any local variables in the parameter expressions then the values of these variables will have to be passed to the point at which the expression is translated and to achieve this effect a lambda expression is wrapped round the expression which instantiates the library block, and the invocation of the specification is complicated by adding an application to the corresponding theta term.

First it is necessary to check that a full set of transmitted parameters is available for instantiation, if this is not the case the match is rejected. Next the parameters must be translated according to their translation codes. Each parameter translation can either suceed, reject the match, or fail. If any one fails, the instantiation fails. If none fail but one or more rejects the match then the instantiation rejects the match. If all succeed, the required details of the instantiation can be constructed.

The details are as follows:

- The definition of the schema for the block instance uses a schema name which is formed from the concatenation of the enclosing system and subsystem block names (from the *PATH*) ended by the name given to the library block instance under consideration.
- The core of the right hand side of this definition will be a simple schema reference if there are no transmitted parameters and no implicitly passed mask variables, otherwise it will be the application of a Z function to a binding display formed by translating each of the actual parameter values which are identified in the *META_ELEMENT* as "Transmitted Parameters", and filling in with any mask variables to be transmitted.
- If there are any local variables either used in parameter expressions or passed as nask variables then the core instantiation of the library block must be enclosed in a lambda expression which is abstracted over the type of bindings grouping together the required set of variable values.
- The declaration required to invoke this definition consisting of a variable name to be declared, which is the blockname in the Simulink diagram, and a type for the variable, which is either a simple reference to the schema defined or an application of the named abstraction to an appropriate theta term, formed using a horizontal schema.
- Since the schema or abstraction name is formed from the blockname by prefixing it with the containing system and subsystem names it will be different from the blockname unless these names are empty. Empty system and blocknames are therefore not supported.

z

$$\boldsymbol{vars2udecs}: \mathbb{F}\ PVALUE \rightarrow seq\ Z\_DEC$$

$$\begin{array}{l}
\forall\ vars: \mathbb{F}\ PVALUE \bullet \\
vars2udecs\ vars = \\
(\mu\ decs: seq\ Z\_DEC \\
\mid ran\ decs = \\
\qquad \{v{:}vars \\
\qquad \bullet\ DecDec\ (\\
\qquad\qquad names \mathrel{\widehat{=}} \langle pvalue2ident\ v\rangle, \\
\qquad\qquad type \mathrel{\widehat{=}} Ident\ Ui)\})
\end{array}$$

z

$$\boldsymbol{vars2uhschem}: \mathbb{F}\ PVALUE \rightarrow Z\_EXPR$$

$$\begin{array}{l}
\forall\ vars: \mathbb{F}\ PVALUE \bullet \\
vars2uhschem\ vars = ZHSchema(\\
\qquad\qquad decl \mathrel{\widehat{=}} vars2udecs\ vars, \\
\qquad\qquad pred \mathrel{\widehat{=}} PredConj\{\})
\end{array}$$

The following function, give a set of variable names and a *Z_EXPR* returns the *Z_EXPR* which consists of a lambda expression of which the declaration part is a horizontal schema whose components are the names supplied and the body is the supplied *Z_EXPR*.

z

$$\boldsymbol{make\_hschema\_abstraction}: \mathbb{F}\ PVALUE \rightarrow Z\_EXPR \rightarrow Z\_EXPR$$

$$\begin{array}{l}
\forall\ vars: \mathbb{F}\ PVALUE;\ ze: Z\_EXPR \bullet \\
make\_hschema\_abstraction\ vars\ ze = \\
ZLambdaExp\ (\\
\qquad decl \mathrel{\widehat=} \langle DecSchema(vars2uhschem\ vars) \rangle, \\
\qquad exp \mathrel{\widehat=} ze)
\end{array}$$

The following function defines the specifications generated to invoke a library block. Since the introduction of support for action subsystems there may be up to three library specifications for each variant of a library block, one giving the normal function and one each for helding or resetting the state, if there is any. Each of these is instantiated in exactly the same way, they are required to have the same parameterization, though each has a distinct name in the library (all names are in the metadata) and each is given a distinct name after instantiation, in this case by use of a suffixing convention.

The following two functions, which create a specification and an invocation (i.e. a declaration to be used when referring to the specification) each address one of the possible three schemas which may be in the library. The are supplied with the required names and will be used up to three times when constructing the block info for the library block.

The parameters are:

1. the name to be used for the schema being defined
2. the name of the library definition to be instantiated
3. the information about parameter values necessary to construct a binding display passing their values
4. the set of local variables which occur in the parameter expressions (vars1)
5. the set of local variables which must be passed to the schema to be invoked (vars2)

Support for implicit passing of local variables has been introduced for use in block references to subsystems in libraries, but there is no reason in principle why this feature should not also be used in specifications for library blocks invoked in the usual way.

In the following specification of the specification to be produced for instantiating a library block the parameters are:

1. the Z name to be defined

2. the name of the Z library specification to be instantiated
3. a binding giving the translated actual values of the transmitted parameters
4. the set of local variables which occur in the actual parameter expressions
5. the set of local variables which are used in the body of the target specification

The specification will consist of an abstraction in which the value required consists of a binding formed from all the local variables in the two sets. the body of the abstraction applies the target specification to a binding which includes all the actual parameter values and any other local variables used in the body of the target (but not transmitted parameters). These latter values are intended to cover block references to masked subsystems translated by ClawZ, but could in principle also be used in hand written library specifications.

z

```
        make_lib_spec: WORD → IDENT → 𝔽 (IDENT × Z_EXPR)
                          → 𝔽 PVALUE → 𝔽 PVALUE → seq Z_PARA
────────────────────────────────────
∀ defname: WORD; z_name: IDENT; binding: 𝔽 (IDENT × Z_EXPR);
        vars1, vars2 : 𝔽 PVALUE •
make_lib_spec defname z_name binding vars1 vars2
=
if vars1 ∪ vars2 = {}
then ⟨SchemaDef(
        name ≙          defname,
        value ≙         if binding = {} then Ident z_name
                        else Application(Ident z_name,
                                BindingDisplay binding) )⟩
else ⟨AbbrevDef(
        ident ≙         word2ident defname,
        value ≙         make_hschema_abstraction (vars1 ∪ vars2) (
                        Application(
                                Ident z_name,
                                BindingDisplay (
                                        {v:vars2• pvalue2ident v ↦ Ident (pvalue2ident v)}
                                        ⊕ binding)) ))⟩
```

The following function defines the manner of invocation of the above specification. Only one set of variables is supplied which is required to be the union of those which are used in the parameters and those which are used in (required by) the library block.

z

```
make_invocation: WORD → IDENT → 𝔽 PVALUE → Z_DEC
────────────────────────────
∀ defname: WORD; localname: IDENT; vars: 𝔽 PVALUE
• make_invocation defname localname vars
= DecDec(
        names ≙ ⟨localname⟩,
        type ≙ if vars = {}
                then Ident (word2ident defname)
                else Application (
                                Ident (word2ident defname),
                                ZBrackets (ZTheta (vars2uhschem vars))))
```

The following function makes a complete specification by invoking *make_lib_spec* up to three times.

z

$$
\begin{array}{l}
\boldsymbol{make\_lib\_specs}: \\
HOLD\_CONTEXT \rightarrow META\_ELEMENT \rightarrow PATH \rightarrow \\
(\mathbb{F}\ IDENT \times Z\_EXPR) \rightarrow (\mathbb{F}\ PVALUE) \rightarrow \\
((seq\ Z\_PARA) \times INVOCATION) \\
\hline
\forall\ hc: HOLD\_CONTEXT;\ \ me: META\_ELEMENT;\ path: PATH; \\
\quad binding: \mathbb{F}\ (IDENT \times Z\_EXPR);\ used\_mvs,\ used\_mvs',\ me\_mvs: \mathbb{F}\ PVALUE; \\
\quad specs: seq\ Z\_PARA;\ invocation: INVOCATION;\ defname: WORD; \\
\quad localname,\ hzname,\ rzname: IDENT; \\
\quad spec,\ spec_h,\ spec_r: seq\ Z\_PARA;\ inv,\ inv_h,\ inv_r: seq\ Z\_DEC \\
\mid defname = path2globw\ path \\
\land\ localname = path2loci\ path \\
\land\ me\_mvs = me.used\_maskvars \\
\land\ used\_mvs' = used\_mvs \cup me\_mvs \\
\land\ spec = make\_lib\_spec\ defname\ me.z\_name\ binding\ used\_mvs\ me\_mvs \\
\land\ inv = \langle make\_invocation\ defname\ localname\ used\_mvs' \rangle \\
\land\ (hc \notin \{HCHeld,\ HCUnknown\} \lor me.held\_z\_name = Nil \Rightarrow \\
\qquad spec_h = \langle\rangle \land inv_h = \langle\rangle) \\
\land\ (hc \in \{HCHeld,\ HCUnknown\} \land me.held\_z\_name = Value\ hzname \Rightarrow \\
\qquad spec_h = make\_lib\_spec\ (w_h\ defname)\ hzname\ binding\ used\_mvs\ me\_mvs \\
\qquad \land\ inv_h = \langle make\_invocation\ (w_h\ defname)\ localname\ used\_mvs' \rangle) \\
\land\ (hc \notin \{HCReset,\ HCUnknown\} \lor me.reset\_z\_name = Nil \Rightarrow \\
\qquad spec_r = \langle\rangle \land inv_r = \langle\rangle) \\
\land\ (hc \in \{HCReset,\ HCUnknown\} \land me.reset\_z\_name = Value\ rzname \Rightarrow \\
\qquad spec_r = make\_lib\_spec\ (w_r\ defname)\ rzname\ binding\ used\_mvs\ me\_mvs \\
\qquad \land\ inv_r = \langle make\_invocation\ (w_r\ defname)\ localname\ used\_mvs' \rangle) \\
\land\ specs = spec \frown spec_h \frown spec_r \\
\land\ invocation = (OtherInv\ localname,\ (inv,\ inv_h,\ inv_r)) \\
\bullet\ make\_lib\_specs\ hc\ me\ path\ binding\ used\_mvs = (specs,\ invocation)
\end{array}
$$

The following specification defines the *BLOCK_INFO* constructed for a library block from the following parameters:

1. the path in the Simulink model to the block being translated
2. the set of local (masked) variables which occur in the parameters to the block
3. the blockname
4. the full set of parameters for the block from the Simulink model
5. a map giving the translations of the transmitted parameters

6. the *META_ELEMENT* against which the block has been matched
7. the *HOLD_CONTEXT* which indicates which specifications should be generated

z

$$
\begin{array}{l}
\boldsymbol{make\_block\_info}: PATH \rightarrow \mathbb{F}\ PVALUE \rightarrow \mathbb{F}\ PARAM \\
\quad \rightarrow (PNAME \mathrel{+\!\!\!\twoheadrightarrow} TRANSLATION\_RESULT) \rightarrow META\_ELEMENT \\
\quad \rightarrow HOLD\_CONTEXT \rightarrow BLOCK\_INFO \\
\hline
\forall\ path: PATH; portval: PVALUE; pars: (\mathbb{F}\ PARAM); me: META\_ELEMENT; \\
\quad hc: HOLD\_CONTEXT; tr\_pars: PNAME \mathrel{+\!\!\!\twoheadrightarrow} TRANSLATION\_RESULT; \\
\quad binding: \mathbb{F}\ (IDENT \times Z\_EXPR); \\
\quad inport\_types, outport\_types: PVALUE \nrightarrow PORT\_TYPE; \\
\quad pparam: PORT\_PARAM; used\_mvs, me\_mvs, used\_mvs': \mathbb{F}\ PVALUE; \\
\quad inv: INVOCATION; spec: seq\ Z\_PARA \\
\mid \quad binding = \{i: IDENT; z: Z\_EXPR; n: PNAME \\
\qquad\qquad \mid (n \mapsto (TMatch\ z)) \in tr\_pars \land i = pname2ident\ n \\
\qquad\qquad \bullet (i, z)\} \\
\quad \land (name \mathrel{\widehat{=}} Ports, value \mathrel{\widehat{=}} portval) \in pars \\
\quad \land (portval \mapsto pparam) \in parse\_ports\_param \\
\quad \land inport\_types = \{pn{:}1 \mathrel{..} pparam.max\_in \bullet num2pvalue\ pn \mapsto UnknownPT\} \\
\qquad \oplus \quad (me.input\_port\_types\ o\ num2pvalue^{\sim}) \\
\quad \land outport\_types = \{pn{:}1 \mathrel{..} pparam.max\_out \bullet num2pvalue\ pn \mapsto UnknownPT\} \\
\qquad \oplus \quad (me.output\_port\_types\ o\ num2pvalue^{\sim}) \\
\quad \land (spec, inv) = make\_lib\_specs\ hc\ me\ path\ binding\ used\_mvs \\
\bullet \quad make\_block\_info\ path\ used\_mvs\ pars\ tr\_pars\ me\ hc = \\
\quad (pars \mathrel{\widehat{=}} pars, input\_port\_types \mathrel{\widehat{=}} inport\_types, \\
\quad output\_port\_types \mathrel{\widehat{=}} outport\_types, \\
\quad invocation \mathrel{\widehat{=}} inv, \\
\quad specification \mathrel{\widehat{=}} spec, \\
\quad virtual \mathrel{\widehat{=}} VInhibit, \\
\quad used\_maskvars \mathrel{\widehat{=}} used\_mvs')
\end{array}
$$

```
z
│ instantiate_lib_block: PATH → 𝔽 PVALUE → (𝔽 PARAM)
│   → HOLD_CONTEXT → META_ELEMENT → INSTANTIATION_RESULT
├──────────────────────────────
│ ∀ path: PATH; portval: PVALUE; pars: (𝔽 PARAM); me: META_ELEMENT;
│   mask_vars, used_maskvars: 𝔽 PVALUE; needed_pars, available_pars: 𝔽 PNAME;
│   tr_pars: PNAME ⇸ TRANSLATION_RESULT × 𝔽 PVALUE;
│   trs: PNAME ⇸ TRANSLATION_RESULT;hc: HOLD_CONTEXT
│ |     needed_pars = {p: PARAM | p ∈ me.transmit_pars • p.name}
│       ∧ available_pars = {p: PARAM | p ∈ pars • p.name}
│       ∧ tr_pars =    {par, tpar: PARAM
│                      | tpar ∈ me.transmit_pars ∧ par ∈ pars
│                        ∧ tpar.name = par.name
│                      • par.name ↦ par_trans2 mask_vars tpar.value par.value}
│       ∧ trs =        first o tr_pars
│       ∧ used_maskvars = ⋃ (ran (second o tr_pars))
│ •     instantiate_lib_block path mask_vars pars hc me =
│       if (path, me.block_path) ∈ path_match
│       then
│        if needed_pars ⊆ available_pars
│        then  if TNoMatch ∈ ran trs then INoMatch
│              else if TFail ∈ ran trs then IFail
│              else IMatch (make_block_info path used_maskvars pars trs me hc)
│        else INoMatch
│       else INoMatch
```

The function *instantiate_last_match* yields the instantiation of the last *META_ELEMENT* of a *META_FILE* which matches a parameter set.

The following specification prescribes that the metafile is to be searched backwards for a match, taking into account the rejection of matches both by the selection criteria and by the translation of parameters. So long as these give no match the search will continue, but if the parameter translation yields a failure the search will fail.

This specification does not propagate a failure, since the primary function of the failure code is to inhibit further attempts to match. If no match is sucessfully instantiated the *INoMatch* result code is returned.

z

$$
\begin{array}{|l}
\quad \boldsymbol{instantiate\_last\_match}: PVALUE \to META\_FILE \to HOLD\_CONTEXT \\
\qquad \to PATH \to \mathbb{F}\ PVALUE \to (\mathbb{F}\ PARAM) \to INSTANTIATION\_RESULT \\
\hline
\forall\ as\_name: PVALUE;\ mf,\ match\_mf: META\_FILE;\ hc: HOLD\_CONTEXT; \\
\quad mask\_vars: \mathbb{F}\ PVALUE;\ pars: (\mathbb{F}\ PARAM);\ path: PATH; \\
\quad instantiate\_fun: META\_ELEMENT \to INSTANTIATION\_RESULT; \\
\quad instantiate\_results,\ \ non\_fail\_results: seq\ INSTANTIATION\_RESULT \\
| \qquad match\_mf = mf \upharpoonright match\ as\_name\ pars \\
\qquad \wedge \quad instantiate\_fun = instantiate\_lib\_block\ path\ mask\_vars\ pars\ hc \\
\qquad \wedge \quad instantiate\_results = instantiate\_fun\ o\ match\_mf \\
\qquad \wedge \quad non\_fail\_results = instantiate\_results \\
\qquad\qquad\qquad \upharpoonright (\{IFail\} \cup (ran\ IMatch)) \\
\bullet \\
\quad instantiate\_last\_match\ as\_name\ mf\ hc\ path\ mask\_vars\ pars = \\
\quad last\ non\_fail\_results
\end{array}
$$

The following four functions define the invocations for the various kinds of supported port blocks.

In order to use *sort_by_invkey* on the declarations, we specify first how to derive *INVOCATION*s from the *PORT_INFO*, then sort them and discard the invocation keys.

z

$$
\begin{array}{|l}
\quad \boldsymbol{inport\_invkey}: PVALUE \mathrel{+\!\!\!\to} INVKEY \\
\hline
\quad \forall\ port: PVALUE;\ ik: INVKEY \bullet \\
\quad port \mapsto ik \in inport\_invkey \\
\quad \Leftrightarrow \\
\quad port = sc2pv\ \texttt{"}trigger\texttt{"} \wedge ik = TriggerInv \\
\quad \vee\ port = sc2pv\ \texttt{"}enable\texttt{"} \wedge ik = EnableInv \\
\quad \vee\ port = ifaction \wedge ik = ActionInv\ 0 \\
\quad \vee\ (\exists\ n: \mathbb{N} \\
\qquad \bullet\ port = num2pvalue\ n \\
\qquad \wedge\ ik = InportInv\ n)
\end{array}
$$

z

$$
\begin{array}{|l}
\quad \boldsymbol{outport\_invkey}: PVALUE \mathrel{+\!\!\!\to} INVKEY \\
\hline
\quad \forall\ port: PVALUE;\ ik: INVKEY \bullet \\
\quad port \mapsto ik \in outport\_invkey \\
\quad \Leftrightarrow (\exists\ n: \mathbb{N} \\
\qquad \bullet\ port = num2pvalue\ n \\
\qquad \wedge\ ik = OutportInv\ n)
\end{array}
$$

z

$$\boldsymbol{inport\_block\_invocation}: PVALUE \nrightarrow INVOCATION$$

$$\begin{array}{l}
\forall\ input\_port: PVALUE;\ invk: INVKEY;\ decl, held, reset: Z\_DECL \bullet \\
input\_port \mapsto (invk, (decl, held, reset)) \in inport\_block\_invocation \\
\Leftrightarrow \\
\quad input\_port \mapsto invk \in inport\_invkey \\
\land\ decl = \langle DecDec\ (names \mathrel{\widehat{=}} \langle inport\_name\ input\_port\rangle, type \mathrel{\widehat{=}} Ident\ Ui)\rangle \\
\land\ held = reset = \langle\rangle
\end{array}$$

z

$$\boldsymbol{outport\_block\_invocation}: PVALUE \nrightarrow INVOCATION$$

$$\begin{array}{l}
\forall\ output\_port: PVALUE;\ invk: INVKEY;\ decl, held, reset: Z\_DECL \bullet \\
output\_port \mapsto (invk, (decl, held, reset)) \in outport\_block\_invocation \\
\Leftrightarrow \\
\quad output\_port \mapsto invk \in outport\_invkey \\
\land\ decl = \langle DecDec\ (names \mathrel{\widehat{=}} \langle outport\_name\ output\_port\rangle, type \mathrel{\widehat{=}} Ident\ Ui)\rangle \\
\land\ held = reset = \langle\rangle
\end{array}$$

z

$$\boldsymbol{action\_block\_invocation}: INVOCATION$$

$$\begin{array}{l}
action\_block\_invocation = \\
\qquad (ActionInv\ 0, \\
\qquad (\langle DecDec\ (names \mathrel{\widehat{=}} \langle pvalue2ident\ ActionQ\rangle, type \mathrel{\widehat{=}} Ident\ Ui)\rangle, \langle\rangle, \langle\rangle))
\end{array}$$

This specification concerns invocations for the numbered action ports which are added to synthesized *Action* blocks.

z

$$\boldsymbol{action\_port\_id}: \mathbb{N}_1 \rightarrow IDENT$$

$$\begin{array}{l}
\forall n: \mathbb{N}_1 \bullet\ action\_port\_id\ n \\
\qquad = pvalue2ident\ (sc2pv\ (\texttt{"}Action\texttt{"} \frown (pv2sc(num2pvalue\ n)) \frown \texttt{"?"}))
\end{array}$$

z

$$\boldsymbol{action\_port\_invocation}: \mathbb{N}_1 \rightarrow INVOCATION$$

$$\begin{array}{l}
\forall n: \mathbb{N}_1 \bullet\ action\_port\_invocation\ n = (ActionInv\ n, (\langle DecDec\ (\\
\qquad names \mathrel{\widehat{=}} \langle action\_port\_id\ n\rangle, \\
\qquad type \mathrel{\widehat{=}} Ident\ Ui)\rangle, \langle\rangle, \langle\rangle))
\end{array}$$

z

$\boldsymbol{action\_port\_invocations}: \mathbb{N}_1 \rightarrow \mathbb{F}\ INVOCATION$

$\forall n: \mathbb{N}_1 \bullet\ action\_port\_invocations\ n =$
$\quad \{m: \mathbb{N}_1 \mid m \leq n \bullet action\_port\_invocation\ m\}$

z

$\boldsymbol{trigger\_block\_invocation}: INVOCATION$

$trigger\_block\_invocation =$
$\quad (TriggerInv,$
$\quad (\langle DecDec\ (names \mathrel{\widehat{=}} \langle pvalue2ident\ TriggerQ\rangle, type \mathrel{\widehat{=}} Ident\ Ui)\rangle, \langle\rangle, \langle\rangle))$

z

$\boldsymbol{enable\_block\_invocation}: INVOCATION$

$enable\_block\_invocation =$
$\quad (EnableInv,$
$\quad (\langle DecDec\ (names \mathrel{\widehat{=}} \langle pvalue2ident\ EnableQ\rangle, type \mathrel{\widehat{=}} Ident\ Ui)\rangle, \langle\rangle, \langle\rangle))$

The following specifies the *block_info* for port blocks.

z

$$
\begin{array}{|l}
\quad \boldsymbol{port\_block\_info}: PVALUE \rightarrow BLOCK\_INFO \rightarrowtail\!\!\!\!\!+ BLOCK\_INFO \\
\hline
\forall\ block\_type: PVALUE;\ block\_info,\ block\_info': BLOCK\_INFO \\
\bullet\ (block\_info \mapsto block\_info') \in port\_block\_info\ block\_type \\
\Leftrightarrow (\exists\ inv: INVOCATION \bullet \\
\qquad (\exists\ port: PVALUE \bullet \\
\qquad (name \mathrel{\widehat{=}} Port,\ value \mathrel{\widehat{=}} port) \in block\_info.pars \\
\qquad \wedge\ (block\_type = InPort \\
\qquad\quad \wedge\ port \mapsto inv \in inport\_block\_invocation \\
\qquad\ \ \vee\ block\_type = OutPort \\
\qquad\quad \wedge\ port \mapsto inv \in outport\_block\_invocation)) \\
\quad\ \ \vee\ block\_type = ActionPort \\
\qquad \wedge\ inv = action\_block\_invocation \\
\quad\ \ \vee\ block\_type = EnablePort \\
\qquad \wedge\ inv = enable\_block\_invocation \\
\quad\ \ \vee\ block\_type = TriggerPort \\
\qquad \wedge\ inv = trigger\_block\_invocation \\
\quad \wedge\ block\_info' = (pars \mathrel{\widehat{=}} block\_info.pars, \\
\qquad\qquad\qquad input\_port\_types \mathrel{\widehat{=}} \{\},\ output\_port\_types \mathrel{\widehat{=}} \{\}, \\
\qquad\qquad\qquad invocation \mathrel{\widehat{=}} inv, \\
\qquad\qquad\qquad specification \mathrel{\widehat{=}} \langle\rangle, \\
\qquad\qquad\qquad virtual \mathrel{\widehat{=}} VInhibit,\ used\_maskvars \mathrel{\widehat{=}} \{\}))
\end{array}
$$

The following specification describes how a *BLOCK_INFO* is updated when a successful library lookup takes place, and also covers the treatment of port blocks, including trigger and enable blocks.

z

$$
\begin{array}{|l}
\quad \boldsymbol{do\_lib\_block}: PVALUE \rightarrow META\_FILE \rightarrow HOLD\_CONTEXT \rightarrow PATH \\
\qquad\quad \rightarrow \mathbb{F}\ PVALUE \rightarrow BLOCK\_INFO \rightarrowtail\!\!\!\!\!+ BLOCK\_INFO \\
\hline
\forall\ as\_name: PVALUE;\ mf: META\_FILE;\ hc: HOLD\_CONTEXT;\ path: PATH; \\
\quad block\_type: PVALUE;\ block\_info,\ block\_info': BLOCK\_INFO;\ mask\_vars: \mathbb{F}\ PVALUE \\
| \qquad (name \mathrel{\widehat{=}} BlockType,\ value \mathrel{\widehat{=}} block\_type) \in block\_info.pars \\
\bullet\ (block\_info \mapsto block\_info') \in do\_lib\_block\ as\_name\ mf\ hc\ path\ mask\_vars \\
\quad \Leftrightarrow \qquad (block\_type \notin port\_block\_types\ \wedge \\
\qquad\quad instantiate\_last\_match\ as\_name\ mf\ hc\ path\ mask\_vars\ block\_info.pars \\
\qquad\qquad = IMatch\ block\_info') \\
\qquad \vee \quad (block\_type \in port\_block\_types\ \wedge \\
\qquad\quad port\_block\_info\ block\_type\ block\_info \\
\qquad\qquad = block\_info')
\end{array}
$$

In the following the required *PORT* must be a destination port. It is assumed that there will be only one line with any given destination. If there is no line or the line name is empty the value “signaln” is used where “n” is the relevant port number.

z

$$\begin{array}{l}
\boldsymbol{line\_name}: ((\mathbb{F}\ LINE) \times PORT) \rightarrow PVALUE \\
\hline
\forall\ lines: \mathbb{F}\ LINE;\ port: PORT \bullet \\
\quad (\exists line: lines \bullet\ port \in line.destinations) \\
\quad \land \quad line\_name\ (lines,\ port) = \\
\quad\quad (\mu\ line: lines;\ name: PVALUE \\
\quad\quad \mid port \in line.destinations \\
\quad\quad \land\ name = force\_signal\_name\ port.port\ line.name \\
\quad\quad \bullet\ name) \\
\lor \quad \neg\ (\exists line: lines \bullet\ port \in line.destinations) \\
\quad \land\ line\_name\ (lines,\ port) = port2signal\ port.port
\end{array}$$

z

$$\begin{array}{l}
\boldsymbol{input\_details\_from\_type}: (\mathbb{F}\ LINE) \rightarrow PORT \\
\qquad\qquad \rightarrow PORT\_TYPE \rightarrow PORT\_DETAILS \\
\hline
\quad \forall\ lines: \mathbb{F}\ LINE;\ port: PORT;\ ptype: PORT\_TYPE \bullet \\
\quad input\_details\_from\_type\ lines\ port\ ptype = \\
\quad\quad (line\_name \mathrel{\widehat{=}} line\_name\ (lines,\ port), \\
\quad\quad port\_type \mathrel{\widehat{=}} ptype)
\end{array}$$

z

$$\begin{array}{l}
\boldsymbol{ipds\_from\_ipts}: (PVALUE \times (\mathbb{F}\ LINE)) \rightarrow (PVALUE \twoheadrightarrow\!\!\!\!\!\!\mapsto PORT\_TYPE) \\
\qquad\qquad \rightarrow (PVALUE \twoheadrightarrow\!\!\!\!\!\!\mapsto PORT\_DETAILS) \\
\hline
\quad \forall\ bname: PVALUE;\ lines: \mathbb{F}\ LINE;\ ipts: PVALUE \twoheadrightarrow\!\!\!\!\!\!\mapsto PORT\_TYPE \bullet \\
\quad ipds\_from\_ipts\ (bname\ ,\ lines)\ ipts = \\
\quad\quad \{pname: PVALUE;\ pt: PORT\_TYPE;\ pd: PORT\_DETAILS;\ port: PORT \\
\quad\quad \mid pname \mapsto pt \in ipts \\
\quad\quad \land\ port = (block \mathrel{\widehat{=}} bname,\ port \mathrel{\widehat{=}} pname) \\
\quad\quad \land\ pt \mapsto pd \in (input\_details\_from\_type\ lines\ port) \\
\quad\quad \bullet\ pname \mapsto pd\}
\end{array}$$

In the case of the output details the line name is irrelevant and is set to *NullString*.

z

$$\boldsymbol{output\_details\_from\_type}: PORT \rightarrow PORT\_TYPE \rightarrow PORT\_DETAILS$$

$$\begin{array}{l}
\forall\ port: PORT;\ ptype: PORT\_TYPE \bullet \\
output\_details\_from\_type\ port\ ptype = \\
\quad (line\_name \mathrel{\widehat{=}} NullString, \\
\quad port\_type \mathrel{\widehat{=}} ptype)
\end{array}$$

z

$$\begin{array}{l}
\boldsymbol{opds\_from\_opts}: PVALUE \rightarrow (PVALUE \mathrel{+\!\!\!\twoheadrightarrow} PORT\_TYPE) \\
\qquad\qquad \rightarrow (PVALUE \mathrel{+\!\!\!\twoheadrightarrow} PORT\_DETAILS)
\end{array}$$

$$\begin{array}{l}
\forall\ bname: PVALUE;\ ipts: PVALUE \mathrel{+\!\!\!\twoheadrightarrow} PORT\_TYPE \bullet \\
opds\_from\_opts\ bname\ ipts = \\
\quad \{pname{:}PVALUE;\ pt: PORT\_TYPE;\ pd: PORT\_DETAILS;\ port: PORT \\
\quad \mid pname \mapsto pt \in ipts \\
\quad \land port = (block \mathrel{\widehat{=}} bname, port \mathrel{\widehat{=}} pname) \\
\quad \land pt \mapsto pd \in (output\_details\_from\_type\ port) \\
\quad \bullet pname \mapsto pd\}
\end{array}$$

The context for this traversal is the combination of the path, the accumulated set of maskvariables, and the set of lines for the current subsystem. The following function updates this context.

z

$$\begin{array}{l}
\boldsymbol{newc\_maskvars}: \\
\quad (PATH \times (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ LINE) \times HOLD\_CONTEXT) \rightarrow \\
\quad A\_SUBSYS \rightarrow PVALUE \rightarrow \\
\quad (PATH \times (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ LINE) \times HOLD\_CONTEXT)
\end{array}$$

$$\begin{array}{l}
\forall\ path, path': PATH;\ maskvars, newmaskvars: \mathbb{F}\ PVALUE;\ lines, lines': \mathbb{F}\ LINE; \\
\quad ass: A\_SUBSYS;\ pv: PVALUE;\ hc, hc': HOLD\_CONTEXT \\
\mid path' = path \frown \langle pv \rangle \\
\land newmaskvars = get\_maskvars\ ass.subsys\_info.subpars \\
\land lines' = ass.subsys\_info.lines \\
\land hc' = ass.subsys\_info.action\_info.held\_context \\
\bullet newc\_maskvars\ (path, maskvars, lines, hc)\ ass\ pv \\
\quad = (path', maskvars \cup newmaskvars, lines', hc')
\end{array}$$

Next we specify the handling of library blocks on this traversal. This involves matching the block against the library.

z

$$\begin{array}{l}
\boldsymbol{libmap\_maskvars}: META\_FILE \to \\
(PATH \times (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ LINE) \times HOLD\_CONTEXT) \to \\
A\_LIB\_BLOCK \to \{0\} \times A\_LIB\_BLOCK \\
\hline
\forall\ mf: META\_FILE;\ path: PATH;\ maskvars: \mathbb{F}\ PVALUE; \\
\quad lines: \mathbb{F}\ LINE;\ hc: HOLD\_CONTEXT;\ alb,\ alb': A\_LIB\_BLOCK; \\
\quad bi,\ bi': BLOCK\_INFO;\ pi,\ pi': PORT\_INFO \\
\mid bi = alb.block\_info \\
\land\ bi' = \qquad if\ bi \in dom(do\_lib\_block\ NullString\ mf\ hc\ path\ maskvars) \\
\qquad\qquad then\ do\_lib\_block\ NullString\ mf\ hc\ path\ maskvars\ bi \\
\qquad\qquad else\ bi \\
\land\ pi' = (input\_port\_details \mathrel{\widehat=} alb.port\_info.input\_port\_details \\
\qquad\qquad \oplus\ ipds\_from\_ipts\ (last\ path,\ lines)\ bi'.input\_port\_types, \\
\qquad output\_port\_details \mathrel{\widehat=} alb.port\_info.output\_port\_details \\
\qquad\qquad \oplus\ opds\_from\_opts\ (last\ path)\ bi'.output\_port\_types) \\
\land\ alb' = \qquad if\ bi \in dom(do\_lib\_block\ NullString\ mf\ hc\ path\ maskvars) \\
\qquad then\ (block\_info \mathrel{\widehat=} bi',\ port\_info \mathrel{\widehat=} pi') \\
\qquad else\ alb \\
\bullet\ libmap\_maskvars\ mf\ (path,\ maskvars,\ lines,\ hc)\ alb = (0,\ alb')
\end{array}$$

Now the handling of subsystems. At this stage little happens, since subsystem translation cannot take place until block synthesis has been attempted on blocks which fail to match against the library. The only thing that happens here is to record the mask variable context, i.e. the set of variable which have been masked by surrounding subsystems. Note that the *specification*, *invocation* and *used_maskvars* fields are assumed to be empty and are not propagated.

z

$$\begin{array}{l}
\boldsymbol{ssmap\_maskvars}: (PATH \times (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ LINE) \times HOLD\_CONTEXT) \\
\qquad \to A\_SUBSYS \times (PVALUE \twoheadrightarrow\!\!\!\!\!\!\shortmid\ \ \{0\}) \to \{0\} \times A\_SUBSYS \\
\hline
\forall\ path: PATH;\ maskvars: \mathbb{F}\ PVALUE;\ lines: \mathbb{F}\ LINE;\ hc: HOLD\_CONTEXT; \\
\quad ass,\ ass': A\_SUBSYS;\ ssi,\ ssi': SUBSYS\_INFO;\ rmap: PVALUE \mathrel{\ooalign{$\to$\cr\hfil$\shortmid$\hfil}} \{0\} \\
\mid ssi = ass.subsys\_info \\
\land\ ssi' = (subpars \mathrel{\widehat=} ssi.subpars,\ syspars \mathrel{\widehat=} ssi.syspars,\ lines \mathrel{\widehat=} ssi.lines, \\
\qquad specification \mathrel{\widehat=} \langle\rangle,\ invocation \mathrel{\widehat=} (NoInv,\ (\langle\rangle,\ \langle\rangle,\ \langle\rangle)), \\
\qquad virtual \mathrel{\widehat=} VUnknown,\ used\_maskvars \mathrel{\widehat=} \{\},\ mv\_ctxt \mathrel{\widehat=} maskvars, \\
\qquad action\_info \mathrel{\widehat=} ssi.action\_info) \\
\land\ ass' = (subsys\_info \mathrel{\widehat=} ssi',\ port\_info \mathrel{\widehat=} ass.port\_info,\ blocks \mathrel{\widehat=} ass.blocks) \\
\bullet\ ssmap\_maskvars\ (path,\ maskvars,\ lines,\ hc)\ (ass,\ rmap) = (0,\ ass')
\end{array}$$

Finally these components are tied together to give a library look-up traversal over an *A_BLOCK*.

z

$$\boldsymbol{library\_lookup}: META\_FILE \rightarrow A\_BLOCK \nrightarrow A\_BLOCK$$

$$\begin{array}{l}
\forall\ mf: META\_FILE;\ ab: A\_BLOCK;\ ass: A\_SUBSYS \\
\mid ab = ASubsys\ ass \\
\bullet\ (0,\ library\_lookup\ mf\ ab) = a\_block\_map\_cr \\
\qquad (libmap\_maskvars\ mf,\ ssmap\_maskvars,\ newc\_maskvars) \\
\qquad (\qquad \langle param\_value\ ass.subsys\_info.syspars\ Name\rangle, \\
\qquad\qquad \{\}, \\
\qquad\qquad ass.subsys\_info.lines, \\
\qquad\qquad ass.subsys\_info.action\_info.held\_context) \\
\qquad ab
\end{array}$$

6.5 Port Type Injection

Information about the types of ports may be supplied in the ClawZ steering file.

6.5.1 Steering File Processing

The ClawZ steering file is to be parsed according to the grammar in section 2.9 and saved as a *STRUCTURE* in *steering_file*. This section defines some processing on the port type information in the steering file.

This function extracts a set of port type assignments from a steeringfile and is parameterised by the name of the relevant structure (i.e. *InPortTypes* or *OutPortTypes*). If any paths occur more than once they are ignored; a warning should be raised.

z

$$
\begin{array}{l}
\textbf{gettypes}: PNAME \to STRUCTURE \to (PATH \mathrel{+\!\!\!\twoheadrightarrow} seq\ PORT\_TYPE) \\
\hline
\forall pn: PNAME; struc : STRUCTURE; strucs: \mathbb{F}\ STRUCTURE; \\
\quad snv: seq\ [name: PNAME; value: VALUE]; ptas : \mathbb{F}\ (PATH \times seq\ PORT\_TYPE) \\
\mid Structure\ snv = struc \\
\land strucs = \{s: STRUCTURE \mid (name \mathrel{\widehat{=}} pn, value \mathrel{\widehat{=}} Struct\ s) \in ran\ snv\} \\
\land ptas = \\
\qquad \{snv2: seq\ [name: PNAME; value: VALUE]; pathn: PNAME; \\
\qquad\ \ types: PVALUE; path: PATH; spt: seq\ PORT\_TYPE \\
\qquad \mid Structure\ snv2 \in strucs \\
\qquad \land (name \mathrel{\widehat{=}} pathn, value \mathrel{\widehat{=}} Simple\ (Qvalue, types)) \in ran\ snv2 \\
\qquad \land pathn \mapsto path \in parse\_pathn \\
\qquad \land types \mapsto spt \in parse\_port\_types \\
\qquad \bullet path \mapsto spt\} \\
\bullet gettypes\ pn\ struc = \\
\qquad \{path : PATH \mid \exists_1\ spt: seq\ PORT\_TYPE \bullet path \mapsto spt \in ptas\} \\
\qquad \lhd ptas
\end{array}
$$

z

$$
\begin{array}{l}
\textbf{inport\_types}: PATH \mathrel{+\!\!\!\twoheadrightarrow} seq\ PORT\_TYPE; \\
\textbf{outport\_types}: PATH \mathrel{+\!\!\!\twoheadrightarrow} seq\ PORT\_TYPE \\
\hline
inport\_types = gettypes\ InPortTypes\ steering\_file \\
\land \quad outport\_types = gettypes\ OutPortTypes\ steering\_file
\end{array}
$$

6.5.2 Model Processing

In this section a pass over the model is specified in which the port type information supplied in the steering file is inserted into the model. The information overrides any type information already in place (e.g. information obtained during library matching and instantiation).

The following updates a port details map from a sequence of port types. The line names are retained from the original, any new ports have no line name associated with them.

z

$$
\begin{array}{l}
\textbf{update\_port\_details}: (seq\ PORT\_TYPE) \to (PVALUE \twoheadrightarrow\!\!\!\!\!\mapsto PORT\_DETAILS) \\
\qquad \to (PVALUE \twoheadrightarrow\!\!\!\!\!\mapsto PORT\_DETAILS) \\
\hline
\forall\ spt : seq\ PORT\_TYPE;\ pdts, pdts' : PVALUE \twoheadrightarrow\!\!\!\!\!\mapsto PORT\_DETAILS \\
\mid pdts' = pdts \oplus \\
\qquad\qquad \{n: dom\ spt;\ pv, name : PVALUE \\
\qquad\qquad \mid pv = num2pvalue\ n \\
\qquad\qquad \land name = \quad if\ pv \in dom\ pdts \\
\qquad\qquad\qquad\qquad then\ (pdts\ pv).line\_name \\
\qquad\qquad\qquad\qquad else\ NullString \\
\qquad\qquad \bullet pv \mapsto (line\_name \mathrel{\widehat=} name, port\_type \mathrel{\widehat=} spt\ n)\} \\
\bullet\ update\_port\_details\ spt\ pdts = pdts'
\end{array}
$$

The following function updates the *PORT_INFO* for a block and returns the new value together with two sets of paths indicating which paths have had their input/ouput ports updated respectively (these will be empty or the singleton set containing the supplied path parameter). The port information is obtained from *inport_types* and *outport_types*.

z

$$
\begin{array}{l}
\textbf{update\_port\_info}: PATH \times PORT\_INFO \rightarrowtail\!\!\!\!\!\mapsto \\
\qquad (\mathbb{F}\ PATH) \times (\mathbb{F}\ PATH) \times PORT\_INFO \\
\hline
\forall\ path: PATH;\ pi, pi': PORT\_INFO;\ pseti, pseto: \mathbb{F}\ PATH; \\
\qquad ipd', opd' : PVALUE \rightarrowtail\!\!\!\!\!\mapsto PORT\_DETAILS \\
\mid (ipd', pseti) = \\
\qquad if\ path \in dom\ inport\_types \\
\qquad then\ (update\_port\_details\ (inport\_types\ path)\ pi.input\_port\_details, \{path\}) \\
\qquad else\ (pi.input\_port\_details, \{\}) \\
\land (opd', pseto) = \\
\quad if\ path \in dom\ outport\_types \\
\quad then\ (update\_port\_details\ (outport\_types\ path)\ pi.output\_port\_details, \{path\}) \\
\quad else\ (pi.output\_port\_details, \{\}) \\
\land pi' = (input\_port\_details \mathrel{\widehat=} ipd', output\_port\_details \mathrel{\widehat=} opd') \\
\bullet\ update\_port\_info\ (path, pi) = (pseti, pseto, pi')
\end{array}
$$

We are now ready to specify the functions used for the traversal. The update is done in one pass over the model, many references to the port information, assuming that the former is much larger less efficiently accessed than the latter.

The first function defines action on subsystems. Information about which paths have been processed is accumulated. Setting port information on subsystems is supported.

z

$$\begin{array}{l}
\textbf{\textit{ssmap\_pti}}: PATH \to A\_SUBSYS \times (PVALUE \rightarrowtail\!\!\!\!\!\!+ (\mathbb{F}\ PATH) \times (\mathbb{F}\ PATH)) \\
\qquad \to ((\mathbb{F}\ PATH) \times (\mathbb{F}\ PATH)) \times A\_SUBSYS \\
\hline
\forall\ as,\ as' : A\_SUBSYS;\ pathmap : PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} (\mathbb{F}\ PATH) \times (\mathbb{F}\ PATH); \\
\quad path : PATH;\ match1,\ match2 : \mathbb{F}\ PATH; \\
\quad port\_info' : PORT\_INFO;\ ipd',\ opd' : PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} PORT\_DETAILS \\
\mid (match1,\ match2,\ port\_info') = update\_port\_info\ (path,\ as.port\_info) \\
\land\ as' = \qquad (subsys\_info \mathrel{\widehat=} as.subsys\_info, \\
\qquad\qquad\qquad port\_info \mathrel{\widehat=} port\_info', \\
\qquad\qquad\qquad blocks \mathrel{\widehat=} as.blocks) \\
\bullet\ ssmap\_pti\ path\ (as,\ pathmap) = \\
\qquad ((\bigcup\ (ran\ (pathmap \mathbin{⨾} first)) \cup match1, \\
\qquad \bigcup\ (ran\ (pathmap \mathbin{⨾} second)) \cup match2), \\
\qquad as')
\end{array}$$

Now we define the action on library blocks. The two path sets in the result indicate what updates if any took place.

z

$$\begin{array}{l}
\textbf{\textit{libmap\_pti}}: PATH \to A\_LIB\_BLOCK \to \\
\qquad ((\mathbb{F}\ PATH) \times (\mathbb{F}\ PATH)) \times A\_LIB\_BLOCK \\
\hline
\forall path : PATH;\ alb,\ alb' : A\_LIB\_BLOCK;\ match1,\ match2 : \mathbb{F}\ PATH; \\
port\_info' : PORT\_INFO;\ ipd',\ opd' : PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} PORT\_DETAILS \\
\mid (match1,\ match2,\ port\_info') = update\_port\_info\ (path,\ alb.port\_info) \\
\land\ alb' = (port\_info \mathrel{\widehat=} port\_info',\ block\_info \mathrel{\widehat=} alb.block\_info) \\
\bullet\ libmap\_pti\ path\ alb = ((match1,\ match2),\ alb')
\end{array}$$

The traversal will work with a path as the context. The following function augments this context.

z

$$\begin{array}{l}
\textbf{\textit{newc\_path}}: PATH \to A\_SUBSYS \to PVALUE \to PATH \\
\hline
\forall\ p : PATH;\ ass : A\_SUBSYS;\ pv : PVALUE \bullet \\
newc\_path\ p\ ass\ pv = p \frown \langle pv \rangle
\end{array}$$

The following function determines the set of paths mentioned in the port type information in the steering file which were not found in the pass over the model.

z

$$\boldsymbol{incorrect\_paths}: (\mathbb{F}\ PATH) \times (\mathbb{F}\ PATH) \rightarrow (\mathbb{F}\ PATH) \times (\mathbb{F}\ PATH)$$

$$\begin{array}{l}
\forall ip,\ ip',\ op,\ op' : \mathbb{F}\ PATH \\
\mid op' = dom\ outport\_types \setminus op \\
\land\ ip' = dom\ inport\_types \setminus ip \\
\bullet\ incorrect\_paths\ (ip,\ op) = (ip',\ op')
\end{array}$$

We now specify a pass over the model which applies the port type information in the steering file. Incorrect path information is evaluated but not returned. It is required that this information be reported in a warning message.

z

$$\boldsymbol{set\_port\_types}: A\_BLOCK \rightarrow A\_BLOCK$$

$$\begin{array}{l}
\forall ab,\ ab' : A\_BLOCK;\ ass : A\_SUBSYS; \\
\quad ip,\ ip',\ op,\ op' : \mathbb{F}\ PATH \\
\mid ab = ASubsys\ ass \\
\land\ ((ip,\ op),\ ab') = a\_block\_map\_cr \\
\qquad\qquad (libmap\_pti,\ ssmap\_pti,\ newc\_path) \\
\qquad\qquad (\langle param\_value\ ass.subsys\_info.syspars\ Name\rangle) \\
\qquad\qquad ab \\
\land\ (ip',\ op') = incorrect\_paths\ (ip,\ op) \\
\bullet\ set\_port\_types\ ab = ab'
\end{array}$$

6.6 Signal Analysis

Signal Analysis propagates information about signals from output ports along lines to inputs ports, and then infers output port information using “hard-coded” knowledge of Mux/Demux and bus constructor/selector blocks. This process is iterated until no more information can be derived, in preparation for synthesis of virtual blocks.

6.6.1 Port Type Specificity

A well-founded relation over *PORT_DETAILS* will be needed to ensure that signal type propagation terminates.

This version of the ordering is written on the assumption that *BusPT* items will not be produced with unknown length (specified as zero), but will only be produced when the width of all the constituent signals is known. If this were later to be changed the ordering would have to be changed.

z

$$\begin{array}{|l}
\boldsymbol{more\_specific\_type}: PORT\_TYPE \leftrightarrow PORT\_TYPE \\
\hline
more\_specific\_type = (\\
\{(UnknownPT,\ GenericPT), \\
\quad (ScalarPT,\ UnknownPT), \\
\quad (VectorPT\ 0,\ UnknownPT)\} \\
\cup \{n : \mathbb{N} \mid n > 0 \bullet (VectorPT\ n,\ VectorPT\ 0)\} \\
\cup \{n : \mathbb{N};\ si: seq\ PORT\_DETAILS \bullet (BusPT\ (n,\ si),\ VectorPT\ n)\} \\
)^{+}
\end{array}$$

The ordering is lifted to *PORT_DETAILS* as follows.

z

$$\begin{array}{|l}
\boldsymbol{more\_specific\_details}: PORT\_DETAILS \leftrightarrow PORT\_DETAILS \\
\hline
more\_specific\_details = \\
\quad \{pd1,\ pd2: PORT\_DETAILS \\
\quad \mid (pd1.port\_type,\ pd2.port\_type) \in more\_specific\_type\}
\end{array}$$

6.6.2 Propagation Over Lines

In the following, remember that a *PORT* by itself may be ambiguous since standard ports are given numeric values both for input and for output ports, and the only way to know whether a port is an input or an output port is by reference to the context (source or destination) in which it occurs in a *LINE*. In all the following definitions involving port values as inputs it is essential that it is known whether the relevant ports are input (destination) or output (source) ports, and that these are not mixed together.

The following function derives a port-to-port relation for use in signal propagation. Note that the *PORT*s in the domain are input/destination ports and those in the codomain output/source ports. Since a line can have only one source, and no two lines can connect to the same destination, the line map is a function

z

$$\boldsymbol{LINE\_MAP} \mathrel{\widehat{=}} PORT \twoheadrightarrow\!\!\!\!\!\!\!+\ PORT$$

z

$$
\begin{array}{|l}
\boldsymbol{lines\_to\_map}: \mathbb{F}\ LINE \rightarrow LINE\_MAP \\
\hline
\forall\ lines: \mathbb{F}\ LINE \bullet \\
lines\_to\_map\ lines = \\
\qquad \{inport,\ outport: PORT;\ line: lines \\
\qquad \mid inport \in line.destinations \\
\qquad \land\ outport = line.source \\
\qquad \bullet\ inport \mapsto outport\}
\end{array}
$$

This function takes a line map and a set of values for output ports, returning the resulting set of values for input ports (assuming that the line map maps input (destination) ports to output (source) ports).

z

$$
\begin{array}{|l}
\boldsymbol{map\_along\_lines}: LINE\_MAP \rightarrow (PORT \nrightarrow PORT\_DETAILS) \\
\qquad \rightarrow (PORT \nrightarrow PORT\_DETAILS) \\
\hline
\forall\ line\_map: LINE\_MAP;\ outport\_map: PORT \nrightarrow PORT\_DETAILS \bullet \\
map\_along\_lines\ line\_map\ outport\_map = line\_map \fatsemi outport\_map
\end{array}
$$

The following function updates an *A_BLOCK* from a set of new *PORT_DETAILS* for one or more of its input ports. It checks whether a prospective update to the details of a port on an *A_BLOCK* are material and admissible. They are material if they differ from the current details, and admissible if more specific than the current details, both of these are checked using *more_specific_details*.

Note that in updating the *PORT_DETAILS* the line name is left unchanged. This is because line names are correct on input ports but set to *NullString* on output ports, as set up by the library lookup procedures.

z

$$\begin{array}{l}
\textbf{\textit{update\_block\_input\_ports}}: ((PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} PORT\_DETAILS) \times A\_BLOCK) \\
\qquad\qquad\qquad\qquad \mathrel{+\!\!\!\!\rightarrow} A\_BLOCK \\
\hline
\forall\ new\_detail\_map: PVALUE \mathrel{+\!\!\!\!\rightarrow} PORT\_DETAILS;\ ab, ab': A\_BLOCK \bullet \\
(new\_detail\_map, ab) \mapsto ab' \in update\_block\_input\_ports \\
\Leftrightarrow \\
(\exists\ si: SUBSYS\_INFO;\ pi, pi': PORT\_INFO;\ bi: BLOCK\_INFO;\ port: PVALUE; \\
\ old\_detail\_map: PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} PORT\_DETAILS;\ blocks: PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} A\_BLOCK; \\
\ good\_detail\_changes: PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} PORT\_DETAILS \\
\bullet\ (\quad ab\ = ALibBlock\ (block\_info \mathrel{\widehat{=}} bi, port\_info \mathrel{\widehat{=}} pi) \\
\qquad \land\ ab' = ALibBlock\ (block\_info \mathrel{\widehat{=}} bi, port\_info \mathrel{\widehat{=}} pi') \\
\quad \lor\quad ab\ = ASubsys\ (subsys\_info \mathrel{\widehat{=}} si, port\_info \mathrel{\widehat{=}} pi, blocks \mathrel{\widehat{=}} blocks) \\
\qquad \land\ ab' = ASubsys\ (subsys\_info \mathrel{\widehat{=}} si, port\_info \mathrel{\widehat{=}} pi', blocks \mathrel{\widehat{=}} blocks) \\
\quad) \\
\land\ old\_detail\_map = pi.input\_port\_details \\
\land\ good\_detail\_changes = \\
\qquad \{pv: PVALUE;\ new\_detail, new\_detail', old\_detail: PORT\_DETAILS \\
\qquad \mid pv \mapsto old\_detail \in old\_detail\_map \\
\qquad \land\ pv \mapsto new\_detail \in new\_detail\_map \\
\qquad \land\ (new\_detail, old\_detail) \in more\_specific\_details \\
\qquad \land\ new\_detail' = \\
\qquad\qquad (line\_name \mathrel{\widehat{=}} old\_detail.line\_name, \\
\qquad\qquad port\_type \mathrel{\widehat{=}} new\_detail.port\_type) \\
\qquad \bullet\ pv \mapsto new\_detail' \\
\qquad \} \\
\land\ good\_detail\_changes \neq \{\} \\
\land\ pi' = \qquad (input\_port\_details \mathrel{\widehat{=}} old\_detail\_map \oplus good\_detail\_changes, \\
\qquad\qquad output\_port\_details \mathrel{\widehat{=}} pi.output\_port\_details) \\
)
\end{array}$$

The following function is used to apply the results of propagating signal types over the lines in a subsystem. It updates an *A_SUBSYS* from a set of locations of input ports (of blocks in the subsystem) and details. It returns, as well as the new *A_SUBSYS* a set of blocknames for blocks whose inputs were changed in this process.

z

$$\begin{array}{l}
\textbf{\textit{update\_subsys\_input\_ports}}: (PORT \nrightarrow PORT\_DETAILS) \times A\_SUBSYS \\
\quad \rightarrow (A\_SUBSYS \times \mathbb{F}\ PVALUE) \\
\hline
\forall\ pdets: PORT \nrightarrow PORT\_DETAILS;\ as,\ as': A\_SUBSYS;\ pvs: \mathbb{F}\ PVALUE; \\
\quad ssi: SUBSYS\_INFO;\ pi: PORT\_INFO;\ blocks,\ blocks': PVALUE \twoheadrightarrow A\_BLOCK \\
\mid ssi = as.subsys\_info \land pi = as.port\_info \land blocks = as.blocks \\
\land\ blocks' = \quad \{blockname: PVALUE;\ ab,\ ab': A\_BLOCK; \\
\qquad\qquad details: PVALUE \nrightarrow PORT\_DETAILS \\
\qquad \mid details = \quad \{portname: PVALUE;\ p{:}PORT;\ pds: PORT\_DETAILS \\
\qquad\qquad \mid p.block = blockname \\
\qquad\qquad \land\ p.port = portname \\
\qquad\qquad \land\ p \mapsto pds \in pdets \\
\qquad\qquad \bullet\ portname \mapsto pds\} \\
\qquad \land\ blockname \mapsto ab \in blocks \\
\qquad \land\ (details,\ ab) \mapsto ab' \in update\_block\_input\_ports \\
\qquad \bullet\ blockname \mapsto ab'\} \\
\land\ pvs = dom\ blocks' \\
\land\ as' = (subsys\_info \mathrel{\widehat=} ssi,\ port\_info \mathrel{\widehat=} pi,\ blocks \mathrel{\widehat=} blocks \oplus blocks') \\
\bullet\ update\_subsys\_input\_ports\ (pdets,\ as) = (as',\ pvs)
\end{array}$$

The type *LINE_FUN* is introduced for a function which propagates values over a set of lines in a subsystem, returning a new subsystem and a set of blocknames of blocks whose input ports have been changed.

z

$$\begin{array}{l}
\textbf{\textit{LINE\_FUN}} \mathrel{\widehat=} (PORT \nrightarrow PORT\_DETAILS) \\
\quad \rightarrow A\_SUBSYS \rightarrow (A\_SUBSYS \times \mathbb{F}\ PVALUE)
\end{array}$$

The following function propagates values from a set of changed output ports over the lines of a subsystem yielding a new subsystem and a set of blocknames of blocks whose input values have been modified.

z

$$\begin{array}{l}
\textbf{\textit{prop\_over\_lines}}: LINE\_MAP \rightarrow LINE\_FUN \\
\hline
\forall\ line\_map: LINE\_MAP;\ ports: PORT \nrightarrow PORT\_DETAILS;\ as,\ as': A\_SUBSYS \\
\bullet\ prop\_over\_lines\ line\_map\ ports\ as = \\
\quad update\_subsys\_input\_ports\ (line\_map \mathbin{⨾} ports,\ as)
\end{array}$$

6.6.3 Block Propagation Preliminaries

This section contains specification generic to block processing and is followed by sections specific to library and to subsystem blocks.

The type of a block processing function is as follows:

z
$$\boldsymbol{BLOCK\_FUN} \mathrel{\widehat=} \mathbb{F}\ PVALUE \to A\_SUBSYS \\ \quad \to (A\_SUBSYS \times (PORT \nrightarrow PORT\_DETAILS))$$

The update is to be confined to blocks whose names are in the first parameter. The right element of the resulting pair indicates which output ports have changed.

We will need to combine block processing functions which is done as follows:

z
$$\boldsymbol{then\_block\_fun}: BLOCK\_FUN \to BLOCK\_FUN \to BLOCK\_FUN$$

$$\begin{array}{l} \forall bf1,\ bf2: BLOCK\_FUN;\ as,\ as',\ as'': A\_SUBSYS;\ bnames: \mathbb{F}\ PVALUE; \\ \quad pd1,\ pd2: PORT \nrightarrow PORT\_DETAILS \\ \mid (as',\ pd1) = bf1\ bnames\ as \\ \land (as'',\ pd2) = bf2\ bnames\ as' \\ \bullet\ then\_block\_fun\ bf1\ bf2\ bnames\ as = (as'',\ pd1 \oplus pd2) \end{array}$$

Note that this is not really sequential, and is only used to combine the effects of updating the library block and subsystem block outputs. It is intended for use only where the domains of *pd1* and *pd2* are disjoint, so that the override can be implemented as union, or as list concatenation.

6.6.4 Propagation Through Library Blocks

The following function performs an update on the output port details for a block. It takes the original port details, the new details and the block name, and returns the updated details together with port update information for the next step of iteration.

z

$$
\begin{array}{l}
\boldsymbol{update\_output\_ports}: ((PVALUE \pfun PORT\_DETAILS) \\
\quad \times (PVALUE \pfun PORT\_DETAILS) \times PVALUE) \\
\quad \to ((PVALUE \pfun PORT\_DETAILS) \times (PORT \pfun PORT\_DETAILS)) \\
\hline
\forall\ old\_detail\_map, new\_detail\_map, good\_changes, results: PVALUE \pfun PORT\_DETAILS; \\
\quad blockname: PVALUE; updates: PORT \pfun PORT\_DETAILS \\
\mid good\_changes = \\
\quad \{pv: PVALUE; new\_detail, old\_detail: PORT\_DETAILS \\
\quad \mid pv \mapsto new\_detail \in new\_detail\_map \\
\quad \land (pv \mapsto old\_detail \in old\_detail\_map \\
\quad\quad \Rightarrow (new\_detail, old\_detail) \in more\_specific\_details) \\
\quad \bullet pv \mapsto new\_detail \\
\quad \} \\
\land results = old\_detail\_map \oplus good\_changes \\
\land updates = \{port: PORT; details: PORT\_DETAILS; portname: PVALUE \\
\quad\quad \mid portname \mapsto details \in good\_changes \\
\quad\quad \land port = (block \mathrel{\widehat=} blockname, port \mathrel{\widehat=} portname) \\
\quad\quad \bullet port \mapsto details\} \\
\bullet update\_output\_ports(old\_detail\_map, new\_detail\_map, blockname) = (results, updates)
\end{array}
$$

The following functions pull together knowledge of how signals propagate through non-subsystem blocks, and use it to update the output ports.

The following type is of functions which are give a blockname and a library block. They then use the information on input ports for this block to infer output types, modifying the output port details and returning information about which output ports had their details changed.

z

$$
\begin{array}{l}
\boldsymbol{LIB\_BLOCK\_FUN} \mathrel{\widehat=} (PVALUE \times A\_LIB\_BLOCK) \\
\quad \pfun ((PVALUE \times A\_LIB\_BLOCK) \times (PORT \pfun PORT\_DETAILS))
\end{array}
$$

The following is the type of a function which knows only about the inference from input types to output types for a single type of library block, and does not know how to update the library block representation. The function is therefore given just the block parameters and the input port details, and returns the inferred output port details (which may or may not be different to the previous output port details).

A function will fail to return a value either because it does not support the particular variant of its block type which has been supplied to it, or because there is insufficient information about the input port types. In the latter case the function might succeed when better port type information becomes available. It is not expected to check the block type parameter.

z

$$\boldsymbol{LIB\_BLOCK\_FUN2} \mathrel{\widehat=} \mathbb{F}\ PARAM \to (PVALUE \leftrightarrow PORT\_DETAILS) \\ \quad \leftrightarrow (PVALUE \leftrightarrow PORT\_DETAILS)$$

We now specify how a *LIB_BLOCK_FUN2* can be converted into a *LIB_BLOCK_FUN*.

z

$$\boldsymbol{lift\_lbf}: LIB\_BLOCK\_FUN2 \to LIB\_BLOCK\_FUN$$

$$\begin{array}{l}
\forall lbf2: LIB\_BLOCK\_FUN2;\ bname: PVALUE;\ alb,\ alb': A\_LIB\_BLOCK; \\
\qquad updates: PORT \leftrightarrow PORT\_DETAILS \bullet \\
\quad (bname,\ alb) \mapsto ((bname,\ alb'),\ updates) \in lift\_lbf\ lbf2 \\
\Leftrightarrow \\
\quad (\exists\ indets,\ outdets,\ oldoutdets,\ newoutdets: PVALUE \leftrightarrow PORT\_DETAILS \\
\quad \bullet\ indets = alb.port\_info.input\_port\_details \\
\quad \land\ (indets \mapsto outdets) \in lbf2\ \ alb.block\_info.pars \\
\quad \land\ oldoutdets = alb.port\_info.output\_port\_details \\
\quad \land\ (newoutdets,\ updates) = update\_output\_ports\ (oldoutdets,\ outdets,\ bname) \\
\quad \land\ alb' = (block\_info \mathrel{\widehat=} alb.block\_info,\ port\_info \mathrel{\widehat=} \\
\qquad\qquad (input\_port\_details \mathrel{\widehat=} alb.port\_info.input\_port\_details, \\
\qquad\qquad output\_port\_details \mathrel{\widehat=} newoutdets)) \\
\quad)
\end{array}$$

In calculating the width of a bus it is necessary to add together the width of all the signals which are included in the bus. The following specifies the sum of a sequence of signal widths.

z

$$\boldsymbol{nat\_seq\_sum}: seq\ \mathbb{N} \to \mathbb{N}$$

$$\begin{array}{ll}
 & nat\_seq\_sum\ \langle\rangle = 0 \\
\land & (\forall n: \mathbb{N};\ ns: seq\ \mathbb{N} \bullet nat\_seq\_sum\ (\langle n\rangle \frown ns) = n + nat\_seq\_sum\ ns)
\end{array}$$

In the following, which specifies the width of a signal of some given *PORT_TYPE*, the value zero should be read “unknown”.

z

$$\boldsymbol{pt2\_width}: PORT\_TYPE \to \mathbb{N}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\forall n: \mathbb{N};\ spd: seq\ PORT\_DETAILS \bullet} \\
 & pt2\_width\ ScalarPT & = 1 \\
\land & pt2\_width\ (VectorPT\ n) & = n \\
\land & pt2\_width\ GenericPT & = 0 \\
\land & pt2\_width\ UnknownPT & = 0 \\
\land & pt2\_width\ (BusPT\ (n,\ spd)) & = n
\end{array}$$

The same convention also applies on extracting the width of a *PORT_DETAILS*.

z

$$\boldsymbol{pd\_width}: PORT\_DETAILS \rightarrow \mathbb{N}$$

$$\forall\ pd: PORT\_DETAILS \bullet$$
$$\quad pd\_width\ pd = pt2\_width\ (pd.port\_type)$$

Default type inference for library blocks is to propagate signal types across generic ports. To do this we need to have available the original type information so that we can tell which ports were originally generic, and therefore type *LIB_BLOCK_FUN2* is not good enough. We first define a function which when supplied with the required extra information gives a *LIB_BLOCK_FUN2*, and then we use that to define the required *LIB_BLOCK_FUN*.

In the following specification:

gipns = generic input port numbers

gopns = generic output port numbers

gipts = generic input port types

ubgipts = upper bounds on the generic input port types

lubgipt = least upper bounds on the generic input port types (at most one!)

opds = output port details (for generic output ports)

z

$$\boldsymbol{default\_update\_outputs\_aux}: BLOCK\_INFO \rightarrow LIB\_BLOCK\_FUN2$$

$$\forall\ bi: BLOCK\_INFO;\ pars: \mathbb{F}\ PARAM;\ gipns,\ gopns: \mathbb{F}\ PVALUE;$$
$$\quad ipds,\ opds: PVALUE \nrightarrow PORT\_DETAILS;$$
$$\quad gipts,\ ubgipts,\ lubgipts: \mathbb{F}\ PORT\_TYPE$$
$$\mid gipns = dom\ (bi.input\_port\_types \rhd \{GenericPT\})$$
$$\wedge\ gopns = dom\ (bi.output\_port\_types \rhd \{GenericPT\})$$
$$\wedge\ gipts = ran\ (gipns \lhd (ipds\ ⨾\ (\lambda PORT\_DETAILS \bullet\ port\_type)))$$
$$\wedge\ ubgipts = \{pt{:}PORT\_TYPE \mid \forall pt2{:}gipts \bullet\ (pt,\ pt2) \in more\_specific\_type \vee pt2 = pt\}$$
$$\wedge\ lubgipts = \{pt{:}ubgipts \mid UnknownPT \notin gipts\ \wedge$$
$$\qquad (\forall pt2{:}ubgipts \bullet\ (pt2,\ pt) \in more\_specific\_type \vee pt2 = pt)\}$$
$$\wedge\ opds = gopns \times \{pt:\ lubgipts \bullet\ (line\_name \mathrel{\widehat{=}} NullString,\ port\_type \mathrel{\widehat{=}} pt)\}$$
$$\bullet\ default\_update\_outputs\_aux\ bi\ pars\ ipds = opds$$

z

$$\boldsymbol{default\_update\_outputs}: LIB\_BLOCK\_FUN$$

$$\forall\ pv: PVALUE;\ alb: A\_LIB\_BLOCK \bullet$$
$$default\_update\_outputs\ (pv,\ alb) =$$
$$lift\_lbf\ (default\_update\_outputs\_aux\ alb.block\_info)\ (pv,\ alb)$$

Bus creators can have only one output, and this is assumed here but not checked. The output port details will have only one entry.

It assumed that there are no gaps in the input port numbering, though this is effectively checked (the existential turns out false because *details* is not then a sequence) and no result is returned if it fails.

It is assumed that there are no non-numeric input ports (e.g. Action, Trigger or Enable). If there are then according to this spec they will be ignored if they have positive width and will otherwise inhibit returning a result.

The width of all input ports must be known. There must be a ports parameter. The input ports must be consecutive and match the number in the ports parameter. Full details of the output port type will then be supplied.

z

$$\boldsymbol{update\_bus\_creator\_outputs}: LIB\_BLOCK\_FUN2$$

$$\forall params: \mathbb{F}\ PARAM;\ ipds,\ opds: PVALUE \mathrel{+\!\!\!\!\rightarrow} PORT\_DETAILS \bullet$$
$$ipds \mapsto opds \in update\_bus\_creator\_outputs\ params$$
$$\Leftrightarrow$$
$$(\exists\ pt2: PORT\_TYPE;\ details: seq\ PORT\_DETAILS;$$
$$\quad n: \mathbb{N};\ param: PARAM;\ pp: PORT\_PARAM$$
$$|\ param.name = Ports$$
$$\land\ param.value \mapsto pp \in parse\_ports\_param$$
$$\land\ dom\ ipds = num2pvalue\ (\!|1 .. pp.max\_in|\!)$$
$$\land\ details =\ (id(1\ ..\ pp.max\_in))\ \text{⨾}\ num2pvalue\ \text{⨾}\ ipds$$
$$\land\ n = nat\_seq\_sum\ (details\ \text{⨾}\ pd\_width)$$
$$\land\ pt2 = BusPT\ (n,\ details)$$
$$\bullet\ opds = \{One \mapsto (line\_name \mathrel{\widehat{=}} NullString,\ port\_type \mathrel{\widehat{=}} pt2)\})$$

The following function follows an element of the *OutputSignals* parameter to a *BusSelector* block through the port details of the input port to select the port details for the corresponding output port. You only get an answer if the *PORT_DETAILS* is a bus or the selector is an empty sequence (which will never happen on the *BusSelector* block but happens as a result of recursion of this function).

There is now provision to accept selectors of the form "signaln" for the nth position in any bus even if that is not the name of the nth signal in the bus.

z

$$\mathbf{select\_from\_bus}: PORT\_DETAILS \rightarrow seq\ PVALUE \nrightarrow PORT\_DETAILS$$

$$\begin{array}{l}
\forall pds, pds': PORT\_DETAILS; spv: seq\ PVALUE \bullet \\
\quad spv \mapsto pds' \in select\_from\_bus\ pds \\
\Leftrightarrow \\
\quad spv = \langle\rangle \land pds' = pds \\
\ \lor \\
\quad (\exists\ width, port: \mathbb{N}; spd: seq\ PORT\_DETAILS; pv: PVALUE; pds: PORT\_DETAILS \\
\quad\ \mid pds.port\_type = BusPT\ (width, spd) \\
\quad\ \land port \mapsto pds \in spd \\
\quad\ \land (pds.line\_name = head\ spv \\
\quad\quad \lor \neg(\exists pd: ran\ spd \bullet pd.line\_name = head\ spv) \\
\quad\quad\ \land head\ spv = port2signal\ (num2pvalue\ port)) \\
\quad\ \bullet (tail\ spv) \mapsto pds' \in select\_from\_bus\ pds)
\end{array}$$

We require that a bus selector has exactly one input. If it has muxed output then it also has exactly one output, otherwise it has the same number of outputs as the number of entries in the *OutputSignals* parameter. The following specification checks that there is one element in the input port details and delivers details for an appropriate number of output ports, but does not actually check that the ports match the port details.

z

$$
\begin{array}{|l}
\quad \textbf{\textit{update\_bus\_selector\_outputs}}: LIB\_BLOCK\_FUN2 \\
\hline
\quad \forall params: \mathbb{F}\ PARAM;\ ipds,\ opds: PVALUE \rightarrowtail PORT\_DETAILS \bullet \\
\quad ipds \mapsto opds \in update\_bus\_selector\_outputs\ params \\
\quad \Leftrightarrow \\
\quad dom\ ipds = \{One\} \\
\land \quad (\exists\ ospar,\ muxpar: params;\ osp: OUTPUTSIGNALS\_PARAM; \\
\qquad\quad ibusdets: seq\ PORT\_DETAILS; \\
\qquad\quad outdets: seq\ PORT\_DETAILS;\ inwidth,\ outwidth: \mathbb{N} \\
\quad \bullet\ BusPT\ (inwidth,\ ibusdets) = (ipds\ One).port\_type \\
\quad \land\ inwidth > 0 \\
\quad \land\ ospar.name = sc2pn\ \text{"}OutputSignals\text{"} \\
\quad \land\ ospar.value \mapsto osp \in parse\_outputsignals\_param \\
\quad \land\ outdets = osp\ \fatsemi\ (select\_from\_bus\ (ipds\ One)) \\
\quad \land\ muxpar.name = sc2pn\ \text{"}MuxedOutput\text{"} \\
\quad \land\ (\quad muxpar.value = on \\
\qquad\quad \land\ outwidth = nat\_seq\_sum\ (outdets\ \fatsemi\ pd\_width) \\
\qquad\quad \land\ opds = \{One \mapsto (\quad line\_name \mathrel{\widehat{=}} NullString, \\
\qquad\qquad\qquad\qquad port\_type \mathrel{\widehat{=}} BusPT\ (outwidth,\ outdets))\} \\
\quad\ \ \lor \quad muxpar.value = off \\
\qquad\quad \land\ num2pvalue\ \fatsemi\ opds = outdets) \\
)
\end{array}
$$

For present purposes a *Mux* block is the same as a *BusCreator*.

z

$$
\begin{array}{|l}
\quad \textbf{\textit{update\_mux\_outputs}}: LIB\_BLOCK\_FUN2 \\
\hline
\quad update\_mux\_outputs = update\_bus\_creator\_outputs
\end{array}
$$

This auxiliary function is used to determine the width of an output vector from a demux block with a scalar *Outputs* parameter. The signals are distributed among the output ports in as well balanced a way as possible. The function is supplied the width of the input port and the number of ouput ports as a pair, and the the output port number whose width is required.

z

$$\textbf{\textit{output\_width}}: (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N} \rightarrow \mathbb{N}$$

```
∀ ipw, nop, pn: ℕ •
output_width (ipw, nop) pn
=       if pn <1 then 0
        else if pn > nop then 0
        else ipw div nop +
                if pn ≤ ipw mod nop then 1
                else 0
```

The following specification deduces output port details for a *Demux* block not in bus selection mode, given the type of the input port and the *Outputs* parameter. It may also be used for a *Demux* block in bus selection mode with a vector *Outputs* parameter. Though the Simulink Help for Demux says it only works on vectors, this is not strictly the case, vector-like buses are also accepted, and since all the buses supported by ClawZ are vector-like this function works with all buses, provided that the outputs selection is consistent with the bus width. (i.e. it treats a bus just the same as a vector of the same width). If a *Demux* is used on a bus then all the signal name information is lost, and even if a subbus is output intact it cannot be understood by bus selector block (it will be a plain vector).

z

$$
\begin{array}{l}
\boldsymbol{analyse\_nonbus\_demux}: (OUTPUTS\_PARAM \times PORT\_TYPE) \\
\qquad\qquad \rightarrowtail\!\!\!\!\!+ \; (PVALUE \rightarrowtail\!\!\!\!\!+ \; PORT\_DETAILS) \\
\hline
\forall\ op: OUTPUTS\_PARAM;\ pt2: PORT\_TYPE; \\
\qquad pvmap: PVALUE \rightarrowtail\!\!\!\!\!+ \; PORT\_DETAILS \bullet \\
(op,\ pt2) \mapsto pvmap \in analyse\_nonbus\_demux \\
\Leftrightarrow \\
(\exists\ iw,\ ow: \mathbb{N};\ ows: seq\ \mathbb{N};\ pdets: seq\ PORT\_DETAILS \\
\mid pt2 = VectorPT\ iw \lor pt2 = BusPT\ (iw,\ pdets) \lor pt2 = ScalarPT \land iw = 1 \\
\bullet\ (op = OPScalar\ ow \\
\qquad \land\ ow \leq iw \\
\qquad \land\ pvmap = \\
\qquad\quad \{pn: 1..ow;\ pt2: PORT\_TYPE \\
\qquad\quad \mid pt2 = (\mu\ w{:}\mathbb{N} \mid w = output\_width\ (iw,\ ow)\ pn \\
\qquad\qquad \bullet\ if\ w = 1\ then\ ScalarPT\ else\ VectorPT\ w) \\
\qquad\quad \bullet\ num2pvalue\ pn \mapsto (line\_name \mathrel{\widehat{=}} NullString,\ port\_type \mathrel{\widehat{=}} pt2)\} \\
\ \lor \\
\ op = OPVector\ ows \land nat\_seq\_sum\ ows = iw \\
\qquad \land\ pvmap = \\
\qquad\quad \{n: dom\ ows;\ pv: PVALUE;\ pt2: PORT\_TYPE \\
\qquad\quad \mid pv = num2pvalue\ n \\
\qquad\quad \land\ pt2 = if\ ows\ n = 1\ then\ ScalarPT\ else\ VectorPT\ (ows\ n) \\
\qquad\quad \bullet\ pv \mapsto (line\_name \mathrel{\widehat{=}} NullString,\ port\_type \mathrel{\widehat{=}} pt2)\} \\
\ ))
\end{array}
$$

Now we turn this into a *LIB_BLOCK_FUN2*. This involves parsing the *Outputs* parameter and checking the *BusSelectionMode* parameter. Strictly speaking we do not support bus selection mode, but since this is the same as non-bus-mode when a vector parameter is supplied we allow this case.

It would probably be easy to do bus selection mode fully, if required.

z

$\boldsymbol{update\_demux\_outputs}: LIB\_BLOCK\_FUN2$

$$
\begin{array}{l}
\forall params: \mathbb{F}\ PARAM;\ ipds,\ opds: PVALUE \nrightarrow PORT\_DETAILS \bullet \\
ipds \mapsto opds \in update\_demux\_outputs\ params \\
\Leftrightarrow \\
dom\ ipds = \{One\} \\
\land\quad (\exists\ opar: params;\ op: OUTPUTS\_PARAM; \\
\qquad isigtype: PORT\_TYPE;\ opvec: seq\ \mathbb{N}; \\
\qquad outdets: seq\ PORT\_DETAILS \\
\bullet\ isigtype = (ipds\ One).port\_type \\
\land\ opar.name = sc2pn\ "Outputs" \\
\land\ opar.value \mapsto op \in parse\_outputs\_param \\
\land\ (\quad (\exists\ bsmpar: params \bullet \\
\qquad\quad bsmpar.name = sc2pn\ "BusSelectionMode" \\
\qquad \land\ bsmpar.value = on) \\
\qquad \land\ op = OPVector\ opvec \\
\qquad \land\ (op,\ isigtype) \mapsto opds \in analyse\_nonbus\_demux \\
\quad \lor\quad \neg(\exists\ bsmpar: params \bullet \\
\qquad\quad bsmpar.name = sc2pn\ "BusSelectionMode" \\
\qquad \land\ bsmpar.value = on) \\
\qquad \land\ (op,\ isigtype) \mapsto opds \in analyse\_nonbus\_demux) \\
)
\end{array}
$$

Some blocks must infer type information from their parameters. The following specification gives the type of a *PARAMETER* as a *PORT_DETAILS*.

z

$\boldsymbol{parameter\_pds}: MVARTYPES \rightarrow PARAMETER \nrightarrow PORT\_DETAILS$

$$
\begin{array}{l}
\forall mvartypes: MVARTYPES;\ param: PARAMETER;\ pds: PORT\_DETAILS \bullet \\
param \mapsto pds \in parameter\_pds\ mvartypes \\
\Leftrightarrow \\
(\exists mvt: MVARTYPE \bullet \\
\quad param \mapsto mvt \in parameter\_type\ mvartypes \\
\land\ pds = \\
\qquad if\ \#mvt = 0 \\
\qquad then\ (line\_name \mathrel{\widehat{=}} NullString,\ port\_type \mathrel{\widehat{=}} ScalarPT) \\
\qquad else\ (line\_name \mathrel{\widehat{=}} NullString,\ port\_type \mathrel{\widehat{=}} VectorPT\ (mvt\_length\ mvt)))
\end{array}
$$

This is a variant which interprets unit vectors as scalars.

z

$$\begin{array}{l}
\boldsymbol{parameter\_pds\_nouv}: MVARTYPES \to PARAMETER \rightarrowtail\!\!\!\!\mapsto PORT\_DETAILS \\
\hline
\forall mvartypes: MVARTYPES; param: PARAMETER; pds: PORT\_DETAILS \bullet \\
param \mapsto pds \in parameter\_pds\_nouv\ mvartypes \\
\Leftrightarrow \\
(\exists mvt: MVARTYPE \bullet \\
\quad param \mapsto mvt \in parameter\_type\ mvartypes \\
\land pds = (line\_name \mathrel{\widehat=} NullString, port\_type \mathrel{\widehat=} \\
\qquad\qquad if\ \#mvt = 0 \\
\qquad\qquad then\ ScalarPT \\
\qquad\qquad else\ if\ mvt\_length\ mvt = 1 \\
\qquad\qquad then\ ScalarPT \\
\qquad\qquad else\ VectorPT\ (mvt\_length\ mvt)))
\end{array}$$

This function obtains a similar result given a set of *PARAM*s and the name of the parameter whose type is required.

z

$$\begin{array}{l}
\boldsymbol{get\_parameter\_details}: MVARTYPES \to \mathbb{F}\ PARAM \\
\qquad\qquad \to PNAME \rightarrowtail\!\!\!\!\mapsto PORT\_DETAILS \\
\hline
\forall mvartypes: MVARTYPES; params: \mathbb{F}\ PARAM; pn: PNAME; \\
\qquad\qquad pds: PORT\_DETAILS \bullet \\
pn \mapsto pds \in get\_parameter\_details\ mvartypes\ params \\
\Leftrightarrow \\
(\exists pv: PVALUE; param, param': PARAMETER; used\_mvars: \mathbb{F}\ PVALUE \\
\bullet (name \mathrel{\widehat=} pn, value \mathrel{\widehat=} pv) \in params \\
\land pv \mapsto param \in parse\_param \\
\land (param', used\_mvars) = change\_parameter\_mlnames\ \{\}\ param \\
\land param' \mapsto pds \in parameter\_pds\ mvartypes)
\end{array}$$

This is a “no unit vector” version of the above

z

$$
\begin{array}{|l}
\quad \boldsymbol{get\_parameter\_details\_nouv}: MVARTYPES \rightarrow \mathbb{F}\ PARAM \\
\qquad\qquad \rightarrow PNAME \nrightarrow PORT\_DETAILS \\
\hline
\quad \forall mvartypes: MVARTYPES;\ params: \mathbb{F}\ PARAM;\ pn: PNAME; \\
\qquad pds: PORT\_DETAILS \bullet \\
\quad pn \mapsto pds \in get\_parameter\_details\_nouv\ mvartypes\ params \\
\quad \Leftrightarrow \\
\quad (\exists pv: PVALUE;\ param,\ param': PARAMETER;\ used\_mvars: \mathbb{F}\ PVALUE \\
\quad \bullet (name \mathrel{\widehat{=}} pn,\ value \mathrel{\widehat{=}} pv) \in params \\
\quad \wedge\ pv \mapsto param \in parse\_param \\
\quad \wedge\ (param',\ used\_mvars) = change\_parameter\_mlnames\ \{\}\ param \\
\quad \wedge\ param' \mapsto pds \in parameter\_pds\_nouv\ mvartypes)
\end{array}
$$

For *Constant*s Type inference will fail unless there is a parameter named “Value” whose type can be ascertained.

z

$$
\begin{array}{|l}
\quad \boldsymbol{update\_constant\_outputs}: MVARTYPES \rightarrow LIB\_BLOCK\_FUN2 \\
\hline
\quad \forall mvartypes: MVARTYPES;\ params: \mathbb{F}\ PARAM; \\
\qquad ipds,\ opds: PVALUE \nrightarrow PORT\_DETAILS \bullet \\
\quad ipds \mapsto opds \in update\_constant\_outputs\ mvartypes\ params \\
\quad \Leftrightarrow \\
\quad (\exists\ pds: PORT\_DETAILS \\
\quad \bullet\ ValuePN \mapsto pds \in get\_parameter\_details\ mvartypes\ params \\
\quad \wedge\ opds = \{One \mapsto pds\} \\
\quad)
\end{array}
$$

For *Unit Delay*s Type inference will fail unless there is a parameter named “X0” whose type can be ascertained.

z

$$
\begin{array}{|l}
\quad \boldsymbol{update\_unit\_delay\_outputs}: MVARTYPES \rightarrow LIB\_BLOCK\_FUN2 \\
\hline
\quad \forall mvartypes: MVARTYPES;\ params: \mathbb{F}\ PARAM; \\
\qquad ipds,\ opds: PVALUE \nrightarrow PORT\_DETAILS \bullet \\
\quad ipds \mapsto opds \in update\_unit\_delay\_outputs\ mvartypes\ params \\
\quad \Leftrightarrow \\
\quad (\exists\ pds: PORT\_DETAILS \\
\quad \bullet\ X0 \mapsto pds \in get\_parameter\_details\ mvartypes\ params \\
\quad \wedge\ opds = \{One \mapsto pds\} \\
\quad)
\end{array}
$$

For *Selector*s Type inference will fail unless there is a parameter named “Elements” whose type can be ascertained. There must be an “InputType” parameter whose value is “Vector” and an “InputSource” parameter whose value is “Internal”.

z

$$\begin{array}{l}
\boldsymbol{update\_selector\_outputs}: MVARTYPES \rightarrow LIB\_BLOCK\_FUN2 \\
\hline
\forall mvartypes: MVARTYPES; params: \mathbb{F}\ PARAM; \\
\quad ipds, opds: PVALUE \nrightarrow PORT\_DETAILS \bullet \\
ipds \mapsto opds \in update\_selector\_outputs\ mvartypes\ params \\
\Leftrightarrow \\
(\exists\ pds: PORT\_DETAILS; n: \mathbb{N} \\
\bullet\ (name \mathrel{\widehat{=}} InputType, value \mathrel{\widehat{=}} Vector) \in params \\
\land\ (name \mathrel{\widehat{=}} ElementSrc, value \mathrel{\widehat{=}} Internal) \in params \\
\land\ Elements \mapsto pds \in get\_parameter\_details\_nouv\ mvartypes\ params \\
\land\ opds = \{One \mapsto pds\} \\
)
\end{array}$$

Type inference for *Merge* requires that the parameter *AllowUnequalInputPortWidths* have value “off”, and that all input ports have the same type and a positive width, in which case that type is transmitted to the output port.

z

$$\begin{array}{l}
\boldsymbol{update\_merge\_outputs}: LIB\_BLOCK\_FUN2 \\
\hline
\forall mvartypes: MVARTYPES; params: \mathbb{F}\ PARAM; \\
\quad ipds, opds: PVALUE \nrightarrow PORT\_DETAILS \bullet \\
ipds \mapsto opds \in update\_merge\_outputs\ params \\
\Leftrightarrow \\
(name \mathrel{\widehat{=}} AllowUnequalInputPortWidths, value \mathrel{\widehat{=}} off) \in params \\
\land\ (\exists\ pt: PORT\_TYPE \\
\quad \bullet\ \{pt\} = \{pds{:}ran\ ipds \bullet\ pds.port\_type\} \\
\quad \land\ pt2\_width\ pt > 0 \\
\quad \land\ opds = \{One \mapsto (line\_name \mathrel{\widehat{=}} NullString, port\_type \mathrel{\widehat{=}} pt)\} \\
\quad)
\end{array}$$

We now specify a table giving the applicable *LIB_BLOCK_FUN* for each block type (if there is one).

z

$$\boldsymbol{update\_outputs\_table}: MVARTYPES \rightarrow PVALUE \twoheadrightarrow LIB\_BLOCK\_FUN$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\forall mvartypes: MVARTYPES \bullet\ update\_outputs\_table\ mvartypes = \{} \\
BusCreator & \mapsto & lift\_lbf\ \ update\_bus\_creator\_outputs, \\
BusSelector & \mapsto & lift\_lbf\ \ update\_bus\_selector\_outputs, \\
Constant & \mapsto & lift\_lbf\ \ (update\_constant\_outputs\ \ mvartypes), \\
Demux & \mapsto & lift\_lbf\ \ update\_demux\_outputs, \\
Mux & \mapsto & lift\_lbf\ \ update\_mux\_outputs, \\
Merge & \mapsto & lift\_lbf\ \ update\_merge\_outputs, \\
Selector & \mapsto & lift\_lbf\ \ (update\_selector\_outputs\ \ mvartypes), \\
UnitDelay & \mapsto & lift\_lbf\ \ (update\_unit\_delay\_outputs\ \ mvartypes) \\
\}
\end{array}
$$

Which is used in this function which updates the outputs of a single library block.

z

$$\boldsymbol{update\_libblock\_outputs}: MVARTYPES \rightarrow LIB\_BLOCK\_FUN$$

$$
\begin{array}{l}
\forall mvartypes: MVARTYPES;\ blockname: PVALUE;\ alb,\ alb': A\_LIB\_BLOCK; \\
\qquad pdets: PORT \nrightarrow PORT\_DETAILS \bullet \\
\ (blockname,\ alb) \mapsto ((blockname,\ alb'),\ pdets) \in update\_libblock\_outputs\ \ mvartypes \\
\Leftrightarrow \\
(\exists\ param: alb.block\_info.pars;\ pname: PVALUE;\ pfun: LIB\_BLOCK\_FUN \\
\mid param = (name \mathrel{\widehat{=}} BlockType,\ value \mathrel{\widehat{=}} pname) \\
\land\ pname \mapsto pfun \in \\
\qquad\qquad \{pn: PVALUE \bullet pn \mapsto default\_update\_outputs\} \\
\qquad\qquad \oplus\ (update\_outputs\_table\ \ mvartypes) \\
\bullet\ (blockname,\ alb) \mapsto ((blockname,\ alb'),\ pdets) \in pfun)
\end{array}
$$

The following specification defines the effect of updating the outputs of a set of library blocks in some subsystem (the names of the blocks to be processed are passed as the first parameter).

z

$$\textbf{\textit{update\_libblocks\_outputs}}: MVARTYPES \rightarrow BLOCK\_FUN$$

$$\begin{array}{l}
\forall mvartypes: MVARTYPES;\ bnames: \mathbb{F}\ PVALUE;\ as,\ as': A\_SUBSYS; \\
\quad pdets: PORT \rightarrow\!\!\!\!\!\!+\ PORT\_DETAILS;\ blocks,\ blocks': PVALUE \twoheadrightarrow\!\!\!\!\!\!+\ A\_BLOCK; \\
\quad updates: \mathbb{P}((PVALUE \times A\_LIB\_BLOCK) \times (PORT \rightarrow\!\!\!\!\!\!+\ PORT\_DETAILS)) \\
\mid updates = \quad \{name{:}bnames;\ alb: A\_LIB\_BLOCK; \\
\qquad\qquad upd: (PVALUE \times A\_LIB\_BLOCK) \times (PORT \rightarrow\!\!\!\!\!\!+\ PORT\_DETAILS) \\
\qquad\quad \mid name \mapsto ALibBlock\ alb \in as.blocks \\
\qquad\quad \land (name,\ alb) \mapsto upd \in update\_libblock\_outputs\ mvartypes \\
\qquad\quad \bullet upd\} \\
\land blocks' = blocks \oplus \\
\qquad\quad \{name: PVALUE;\ alb: A\_LIB\_BLOCK;\ pds: PORT \rightarrow\!\!\!\!\!\!+\ PORT\_DETAILS \\
\qquad\quad \mid ((name,\ alb),\ pds) \in updates \bullet name \mapsto ALibBlock\ alb\} \\
\land pdets = \bigcup \{block: PVALUE \times A\_LIB\_BLOCK;\ pds: PORT \rightarrow\!\!\!\!\!\!+\ PORT\_DETAILS \\
\qquad\quad \mid (block,\ pds) \in updates \\
\qquad\quad \bullet pds\} \\
\land as' = \qquad (subsys\_info \mathrel{\widehat=} as.subsys\_info, \\
\qquad\quad port\_info \mathrel{\widehat=} as.port\_info, \\
\qquad\quad blocks \mathrel{\widehat=} blocks') \\
\bullet\ update\_libblocks\_outputs\ mvartypes\ bnames\ as = (as',\ pdets)
\end{array}$$

6.6.5 Propagation Through Subsystem Blocks

This function extracts from an *A_SUBSYS* the input port details of the internal output port blocks.

z

$$\textbf{\textit{extract\_subsys\_outputs}}: A\_SUBSYS \rightarrow\!\!\!\!\!\!+\ (PVALUE \rightarrow\!\!\!\!\!\!+\ PORT\_DETAILS)$$

$$\begin{array}{l}
\forall\ as: A\_SUBSYS \bullet \\
\quad extract\_subsys\_outputs\ as = \\
\qquad \{pname: PVALUE;\ block{:}A\_LIB\_BLOCK;\ btp{,}bpp: PARAM;\ dts{:}PORT\_DETAILS \\
\qquad \mid pname \mapsto (ALibBlock\ block) \in as.blocks \\
\qquad \land \{btp,\ bpp\} \subseteq block.block\_info.pars \\
\qquad \land btp.name = BlockType \land btp.value = OutPort \\
\qquad \land bpp.name = Port \land bpp.value = pname \\
\qquad \land One \mapsto dts \in block.port\_info.input\_port\_details \\
\qquad \bullet pname \mapsto dts\}
\end{array}$$

This function extracts from an *A_SUBSYS* the output port details of the internal input port blocks.

z

$$
\begin{array}{l}
\textbf{\textit{extract\_subsys\_inputs}}: A\_SUBSYS \rightarrowtail\!\!\!\!\!\!\;\; (PVALUE \rightarrowtail PORT\_DETAILS) \\
\hline
\forall\ as: A\_SUBSYS \bullet \\
\quad extract\_subsys\_inputs\ as = \\
\qquad \{pname: PVALUE;\ block: A\_LIB\_BLOCK;\ btp, bpp: PARAM;\ dts: PORT\_DETAILS \\
\qquad \mid pname \mapsto (ALibBlock\ block) \in as.blocks \\
\qquad \land \{btp,\ bpp\} \subseteq block.block\_info.pars \\
\qquad \land btp.name = BlockType \land btp.value = InPort \\
\qquad \land bpp.name = Port \land bpp.value = pname \\
\qquad \land One \mapsto dts \in block.port\_info.output\_port\_details \\
\qquad \bullet pname \mapsto dts\}
\end{array}
$$

The following function returns details of all the output ports of the blocks in a subsystem. This will be used as a starter for iteration the first time a subsystem is invoked for signal propagation.

z

$$
\begin{array}{l}
\textbf{\textit{all\_subsys\_outports}}: A\_SUBSYS \rightarrow (PORT \rightarrowtail PORT\_DETAILS) \\
\hline
\forall as: A\_SUBSYS \bullet \\
all\_subsys\_outports\ as = \\
\{\ bname, pname: PVALUE;\ alb: A\_LIB\_BLOCK; \\
\quad as2: A\_SUBSYS;\ details: PORT\_DETAILS \\
\mid bname \mapsto ALibBlock\ alb \in as.blocks \\
\quad \land (pname \mapsto details) \in alb.port\_info.output\_port\_details \\
\lor\ bname \mapsto ASubsys\ as2 \in as.blocks \\
\quad \land (pname \mapsto details) \in as2.port\_info.output\_port\_details \\
\bullet (block \mathrel{\widehat{=}} bname,\ port \mathrel{\widehat{=}} pname) \mapsto details\}
\end{array}
$$

The following function pushes down information on the input ports of a subsystem to the output ports of the input port blocks inside the subsystem. Information about which ports were updated is returned for use in initiating propagation of values through the subsystem.

This process detects its first use on a particular subsystem, from the lack of prior *PORT_DETAILS* on the input port blocks, and in that case returns the complete set of library block output port details for propagation. This ensures that all the available port information is brought into play at the start of the process, but that only changes are subsequently propagated.

No check is made for increased specificity here, since that will already have taken place when the port details on the subsystem itself were updated.

z

$$
\begin{array}{l}
\boldsymbol{push\_subsys\_inputs}: A\_SUBSYS \\
\quad \rightarrowtail\!\!\!\!\!+ \ (A\_SUBSYS \times OPT\ [PORT \rightarrowtail\!\!\!\!\!+ PORT\_DETAILS]) \\
\hline
\forall as, as': A\_SUBSYS; pdets: OPT\ [PORT \rightarrowtail\!\!\!\!\!+ PORT\_DETAILS] \\
\bullet\ as \mapsto (as', pdets) \in push\_subsys\_inputs \Leftrightarrow \\
\quad (\exists ipds: PVALUE \rightarrowtail\!\!\!\!\!+ PORT\_DETAILS; initialised\_input\_ports: \mathbb{F}\ PVALUE; \\
\qquad blocks, block\_updates, blocks': PVALUE \twoheadrightarrow\!\!\!\!\!+ A\_BLOCK \\
\quad \bullet\ ipds = as.port\_info.input\_port\_details \\
\quad \land\ blocks = as.blocks \\
\quad \land\ block\_updates = \\
\qquad\qquad \{bname, pname: PVALUE; alb, alb': A\_LIB\_BLOCK; \\
\qquad\qquad\qquad params: \mathbb{F}\ params; details: PORT\_DETAILS \\
\qquad\qquad \mid bname \mapsto (ALibBlock\ alb) \in blocks \\
\qquad\qquad \land\ params = alb.block\_info.pars \\
\qquad\qquad \land\ (name \mathrel{\widehat=} BlockType, value \mathrel{\widehat=} InPort) \in params \\
\qquad\qquad \land\ (name \mathrel{\widehat=} Port, value \mathrel{\widehat=} pname) \in params \\
\qquad\qquad \land\ pname \mapsto details \in ipds \\
\qquad\qquad \land\ One \mapsto details \notin alb.port\_info.output\_port\_details \\
\qquad\qquad \land\ alb' = \quad (block\_info \mathrel{\widehat=} alb.block\_info, \\
\qquad\qquad\qquad\qquad port\_info \mathrel{\widehat=} \\
\qquad\qquad\qquad\qquad\quad (input\_port\_details \mathrel{\widehat=} \{\}, \\
\qquad\qquad\qquad\qquad\quad output\_port\_details \mathrel{\widehat=} \{One \mapsto details\})) \\
\qquad\qquad \bullet\ bname \mapsto (ALibBlock\ alb')\} \\
\quad \land\ blocks' = blocks \oplus block\_updates \\
\quad \land\ as' = (\quad subsys\_info \mathrel{\widehat=} as.subsys\_info, \\
\qquad\qquad port\_info \mathrel{\widehat=} as.port\_info, \\
\qquad\qquad blocks \mathrel{\widehat=} blocks') \\
\quad \land\ initialised\_input\_ports = \\
\qquad\qquad \{bname: PVALUE; alb: A\_LIB\_BLOCK \\
\qquad\qquad \mid bname \mapsto (ALibBlock\ alb) \in blocks \\
\qquad\qquad \land\ (name \mathrel{\widehat=} BlockType, value \mathrel{\widehat=} InPort) \in alb.block\_info.pars \\
\qquad\qquad \land\ alb.port\_info.output\_port\_details \neq \{\} \\
\qquad\qquad \bullet\ bname\} \\
\quad \land\ pdets = \quad if\ initialised\_input\_ports = \{\}\ then\ Nil \\
\qquad\qquad else\ Value \\
\qquad\qquad\qquad \{ port{:}PORT; alb: A\_LIB\_BLOCK; port\_details: PORT\_DETAILS \\
\qquad\qquad\qquad \mid port.block \mapsto (ALibBlock\ alb) \in block\_updates \\
\qquad\qquad\qquad \land\ port.port = One \\
\qquad\qquad\qquad \land\ One \mapsto port\_details \in alb.port\_info.output\_port\_details \\
\qquad\qquad\qquad \bullet\ port \mapsto port\_details\})
\end{array}
$$

This function takes a subsystem which has already been subjected to internal signal propagation and lifts the resulting changed output ports to the next higher level. This involves propagation of *PORT_DETAILS* from the contained port blocks to the subsystem itself, and conversion of the *PORT*s used to index the details to refer to the ports on the subsystem rather than those in the subsystem.

z

$$
\begin{array}{l}
\boldsymbol{lift\_subsys\_outputs}: (PVALUE \times A\_BLOCK) \\
\quad \rightarrowtail ((PVALUE \times A\_BLOCK) \times (PORT \rightarrowtail PORT\_DETAILS)) \\
\hline
\forall\ blockname: PVALUE;\ block,\ block': A\_BLOCK;\ outputs: PORT \rightarrowtail PORT\_DETAILS \bullet \\
(blockname \mapsto block) \mapsto (blockname \mapsto block',\ outputs) \in lift\_subsys\_outputs \\
\Leftrightarrow \\
(\exists\ as,\ as': A\_SUBSYS;\ pi,\ pi': PORT\_INFO; \\
\quad new\_outputs,\ old\_outputs,\ output\_changes: PVALUE \rightarrowtail PORT\_DETAILS \\
\bullet\ block = ASubsys\ as \land block' = ASubsys\ as' \\
\land\ new\_outputs = extract\_subsys\_outputs\ as \\
\land\ old\_outputs = as.port\_info.output\_port\_details \\
\land\ (output\_changes, outputs) = update\_output\_ports(old\_outputs,\ new\_outputs,\ blockname) \\
\land\ pi = as.port\_info \\
\land\ pi' = (output\_port\_details \mathrel{\widehat{=}} pi.output\_port\_details \oplus output\_changes, \\
\qquad input\_port\_details \mathrel{\widehat{=}} pi.input\_port\_details) \\
\land\ as' = (subsys\_info \mathrel{\widehat{=}} as.subsys\_info,\ port\_info \mathrel{\widehat{=}} pi',\ blocks \mathrel{\widehat{=}} as.blocks))
\end{array}
$$

The following function specifies by recursion the iteration which propagates signal values through a subsystem. At each step the result includes information about what has changed, either as a set of output ports (resulting from propagation through blocks) or as a set of blocknames (resulting from propagation along lines), which limits the scope of the update on the next step. When the set becomes empty the iteration terminates. The set will become empty because every time a signal value changes it must become more specific, specificity being a well-founded partial ordering on *PORT_DETAILS* (most specific at bottom of this ordering). It is therefore important to ensure when the line and block propagation is written or amended that it does respect the relevant partial ordering and only makes and registers changes which do make the relevant output port details more specific.

Note that as well as the explicit recursion in this definition which implements the iteration, there will be another level of recursion when a suitable *BLOCK_FUN* is defined later, since the propagation of signals through subsystems will also involve recursion.

z

$$\begin{array}{|l}
\qquad \boldsymbol{iterate\_subsys}: LINE\_FUN \times BLOCK\_FUN \\
\qquad\qquad \rightarrow (A\_SUBSYS \times (PORT \rightarrowtail\!\!\!\!\!\!\!+ \;\; PORT\_DETAILS)) \\
\qquad\qquad \rightarrow A\_SUBSYS \\
\hline
\forall\, lf: LINE\_FUN;\ bf: BLOCK\_FUN;\ as,\ as',\ as'',\ as''': A\_SUBSYS; \\
\quad portdata,\ portdata': (PORT \nrightarrow PORT\_DETAILS);\ pvs: \mathbb{F}\ PVALUE \\
\mid portdata = \{\} \land as''' = as \\
\lor\ portdata \neq \{\} \\
\qquad \land (as', pvs) = lf\ portdata\ as \\
\qquad \land (as'', portdata') = bf\ pvs\ as' \\
\qquad \land as''' = iterate\_subsys\ (lf, bf)\ (as'', portdata') \\
\bullet\ iterate\_subsys\ (lf, bf)\ (as, portdata) = as'''
\end{array}$$

The first time signals are propagated through a subsystem all lines and blocks must be processed, as follows:

z

$$\begin{array}{|l}
\qquad \boldsymbol{first\_iterate\_subsys}: LINE\_FUN \times BLOCK\_FUN \\
\qquad\qquad \rightarrow A\_SUBSYS \rightarrow A\_SUBSYS \\
\hline
\forall\, lf: LINE\_FUN;\ bf: BLOCK\_FUN;\ as,\ as',\ as'',\ as''': A\_SUBSYS; \\
\quad portdata,\ portdata': (PORT \nrightarrow PORT\_DETAILS);\ pvs,\ ipvs: \mathbb{F}\ PVALUE \\
\mid portdata = all\_subsys\_outports\ as \\
\land\ ipvs = dom\ as.blocks \\
\land\ (as', pvs) = lf\ portdata\ as \\
\land\ (as'', portdata') = bf\ ipvs\ as' \\
\land\ as''' = iterate\_subsys\ (lf, bf)\ (as'', portdata') \\
\bullet\ first\_iterate\_subsys\ (lf, bf)\ as = as'''
\end{array}$$

The propagation of signal information through subsystems is then accomplished as follows. This involves pushing the input information into the subsystem, iterating, and then lifting the output information out of the subsystem.

z

$$\begin{array}{l}
\boldsymbol{update\_subsystem\_outputs}: BLOCK\_FUN \to (PVALUE \times A\_SUBSYS) \\
\qquad \nrightarrow ((PVALUE \times A\_SUBSYS) \times (PORT \nrightarrow PORT\_DETAILS)) \\
\hline
\forall block\_fun: BLOCK\_FUN; blockname: PVALUE; as, as': A\_SUBSYS; \\
\qquad pdets: PORT \nrightarrow PORT\_DETAILS \bullet \\
(blockname, as) \mapsto ((blockname, as'), pdets) \in update\_subsystem\_outputs\ block\_fun \\
\Leftrightarrow \\
(\exists\ initial\_pdets: OPT[PORT \nrightarrow PORT\_DETAILS]; line\_fun: LINE\_FUN; \\
\qquad initial\_as, iterated\_as: A\_SUBSYS \\
\bullet\ as \mapsto (initial\_as, initial\_pdets) \in push\_subsys\_inputs \\
\land\ line\_fun = prop\_over\_lines\ (lines\_to\_map\ initial\_as.subsys\_info.lines) \\
\land\ iterated\_as = \\
\qquad if\ \ initial\_pdets = Nil \\
\qquad then\ first\_iterate\_subsys\ (line\_fun, block\_fun)\ initial\_as \\
\qquad else\ iterate\_subsys\ (line\_fun, block\_fun)\ (initial\_as, (Value^{\sim})\ initial\_pdets) \\
\land\ ((blockname \mapsto ASubsys\ as'), pdets) \\
\qquad = lift\_subsys\_outputs\ (blockname \mapsto ASubsys\ iterated\_as))
\end{array}$$

The following function does this for a set of selected subsystems, (the ones whose input port details have just been changed).

z

$$
\begin{array}{l}
\quad \textbf{\textit{update\_subsystems\_outputs}}: MVARTYPES \rightarrow BLOCK\_FUN \\
\hline
\forall mvartypes: MVARTYPES;\ bnames: \mathbb{F}\ PVALUE;\ as,\ as': A\_SUBSYS; \\
\quad pdets: PORT \rightarrowtail\!\!\!\!\!\!\mapsto PORT\_DETAILS;\ blocks,\ blocks': PVALUE \twoheadrightarrow A\_BLOCK; \\
\quad updates: \mathbb{P}((PVALUE \times A\_SUBSYS) \times (PORT \nrightarrow PORT\_DETAILS)) \\
\mid updates = \\
\quad \{name{:}bnames;\ ass: A\_SUBSYS; \\
\qquad upd: (PVALUE \times A\_SUBSYS) \times (PORT \nrightarrow PORT\_DETAILS) \\
\quad \mid name \mapsto ASubsys\ ass \in as.blocks \\
\quad \land (name,\ ass) \mapsto upd \in (update\_subsystem\_outputs \\
\qquad (then\_block\_fun\ (update\_libblocks\_outputs\ mvartypes) \\
\qquad\quad (update\_subsystems\_outputs\ mvartypes))) \\
\quad \bullet\ upd\} \\
\land blocks' = blocks \oplus \\
\qquad \{name: PVALUE;\ ass: A\_SUBSYS;\ pdets: PORT \nrightarrow PORT\_DETAILS \\
\qquad \mid ((name,\ ass),\ pdets) \in updates \bullet name \mapsto ASubsys\ ass\} \\
\land pdets = \bigcup \{block: PVALUE \times A\_SUBSYS;\ pdets: PORT \nrightarrow PORT\_DETAILS \\
\qquad \mid (block,\ pdets) \in updates \\
\qquad \bullet\ pdets\} \\
\land as' = \qquad (subsys\_info \mathrel{\widehat{=}} as.subsys\_info, \\
\qquad port\_info \mathrel{\widehat{=}} as.port\_info, \\
\qquad blocks \mathrel{\widehat{=}} blocks') \\
\bullet\ update\_subsystems\_outputs\ mvartypes\ bnames\ as = (as',\ pdets)
\end{array}
$$

6.6.6 Propagation Through Blocks

Now we package it up for the top level:

z

$$
\begin{array}{l}
\quad \textbf{\textit{propagate\_signal\_details}}: MVARTYPES \rightarrow A\_BLOCK \rightarrow A\_BLOCK \\
\hline
\quad \forall\ mvartypes: MVARTYPES;\ ab,\ ab': A\_BLOCK;\ pv: PVALUE; \\
\qquad as,\ as': A\_SUBSYS;\ pdets: PORT \nrightarrow PORT\_DETAILS \\
\quad \mid ab = ASubsys\ as \land ab' = ASubsys\ as' \\
\quad \land (name \mathrel{\widehat{=}} Name,\ value \mathrel{\widehat{=}} pv) \in as.subsys\_info.syspars \\
\quad \land (pv,\ as) \mapsto ((pv,\ as'),\ pdets) \\
\qquad \in update\_subsystem\_outputs \\
\qquad (then\_block\_fun\ (update\_libblocks\_outputs\ mvartypes) \\
\qquad\quad (update\_subsystems\_outputs\ mvartypes)) \\
\quad \bullet\ propagate\_signal\_details\ mvartypes\ ab = ab'
\end{array}
$$

6.7 Virtualization

This section and section 6.8 cover the capability of ClawZ to translate Simulink blocks without using the ClawZ library. The intention was originally to deal with those blocks called by Simulink "virtual" blocks, which are effectively absorbed by Simulink into the wiring of the model. The first attempt to deal with these blocks independently of the Z library involved generating definitions similar in character to those provided in the library. This approach was adopted to minimise disruption to the style of specification produced and hence to QuinetiQ tools automating the processing of the specifications. This approach to dealing with "virtual" blocks has a scope of applicability which goes beyond the types of block treated as virtual by Simulink and is now called "synthesis" throughout this specification.

A more radical approach to handling virtual blocks has also been thought desirable, and is specified here alongside synthesis. This more radical approach we call "virtualization", and is attempted for certain block types if the flag `virtualize` is set by the ClawZ user. It has a narrower scope than the method used for synthesis, being confined to blocks which are purely functional, and is not suitable for dealing with cycles in the wiring diagram (unless broken by a non-virtualized block). It has the benefit of potentially reducing the size of the resulting specification (at least as measured by the number of components in the subsystem schemas).

6.7.1 Traversal Strategies

Virtualization requires a traversal of the *A_BLOCK* which does not fit within the available traversal methods. This method is not thought likely to be more generally applicable and is therefore custom coded for virtualization. This is because virtualization of a block requires virtualization of any other blocks whose outputs are connected to the input of the block, influencing the order of treatment of blocks in a subsystem, and special measures are necessary to ensure termination which might otherwise be jeopardized by cycles in the wiring diagram.

In order to undertake a complex recursive traversal of our tree-like *A_BLOCK* data structure in a manner which is both reasonably efficient and maximally intelligible some consideration has been given to general principles in these matters, which we take a moment here to discuss.

One general method common in functional programming is to separate a general traversal method from the specifics of a particular traversal, and to implement or specify that method as a higher order function or operator. This operator accepts as parameters functions which ecapsulate the specifics of a particular application, and which do not themselves need to be recursive. The method of traversal can then be understood once separately from the details of the applications, and the details of the applications can then be approached without the additional complexity arising from recursion to achieve traversal. Two general methods of traversal have been specified in section 6.2 and have been used where appropriate in this specification.

For virtualization we have attempted an application specific recursion done in a systematic manner as follows.

For the purposes of explaining the method we introduce some terminology. The core of the specification is a set of functions defined by mutual recursion. These functions are called the "principal functions". In order that the principal functions can be defined in separate Z paragraphs spanning several pages of this document, one of the principle functions is singled out for special treatment. We call this function the "key" function, it must amount to the most generally convenient form of the virtualization functionality. The key function will be defined last, and all other principal functions will be define prior to it as operators which derive their function from the key function, i.e. they take the key function as a parameter. The use of a key function in this way permits a large scale mutual recursion to be presented as a set of operators followed by a single recursive function of moderate or low complexity. Because the principal functions other than the key function are parameterized by the key function they do not strictly participate in the recursion, but we will nevertheless talk *as if* all the principal functions are involved in the recursion.

It is desirable at all stages to separate out as much functionality as possible from the main complex of mutual recursion, i.e. from the principal functions. The largest body of material separated out from the recursion is the aspects of virtualization which are specific to particular Simulink block types (see sections 6.7.2 to 6.7.11). A second body of material consists of auxiliary functions which are used in defining the principal functions but which are not themselves principals (i.e. they need not participate in the recursion, and do not require the key function as a parameter).

6.7.2 Library Block Virtualization Method

In this section we specify the library block specific aspects of virtualization in a way which decouples these specifications from the recursion supporting the traversal.

It is the intention that virtualization of an instance of a library block will make use of information derived from the virtualization of the blocks connected to its input ports. To avoid the recursion which might otherwise arise from this dependence the core block specific aspect of virtualization is undertaken in two stages.

The virtualization function first returns the set of input port names whose values are required in order to complete virtualization, together with a function which will complete the virtualization when supplied with the required values. "Completing the virtualisation" in this context simply means returning an output port expression map and the set of used maskvariables. In this way the library block virtualization exports to its calling function the problem of virtualizing the blocks connected to its input ports.

The latter stage of this process is undertaken by a function having the following type:

z

$$\boldsymbol{LIB\_BLOCK\_VIRT\_FUN1} \mathrel{\widehat=} (PVALUE \twoheadrightarrow\!\!\!\!\!\!+\; Z\_EXPR) \rightarrow VIRTUAL \times \mathbb{F}\ PVALUE$$

which is expected to be specific not only to a particular block type but to the full required context excepting only the input port expression map. The parameter to this function is a map of input port names to *Z_EXPR*s which denote the values on those lines. The result is a *VIRTUAL* (which in the case of successful virtualization will include a similar map of expressions for the output ports) and the set of mask variables which have been used in forming the expressions.

A *LIB_BLOCK_VIRT_FUN1* is obtained from a function which undertakes the first stage of the process, and returns with the function the set of portnames for the input ports whose values are required for virtualization. This first stage function is specific only to a Simulink block type, and is therefore supplied a more substantial collection of information to work on.

The information available to the virtualization function for this purpose is as follows:

- The matlab variable types.
- The *A_LIB_BLOCK* which is to be virtualized.
- The set of mask variables in scope.

and it therefore has the following type:

z

$$\begin{array}{l} \boldsymbol{LIB\_BLOCK\_VIRT\_FUN2} \mathrel{\widehat{=}} \\ \quad MVARTYPES \times A\_LIB\_BLOCK \times (\mathbb{F}\ PVALUE) \\ \quad \rightarrowtail LIB\_BLOCK\_VIRT\_FUN1 \times (\mathbb{F}\ PVALUE) \end{array}$$

in which the set of *PVALUE*s returned is the set of input portnames whose values are required to complete virtualization.

As in signal inference and block synthesis, the collection of these functions for each supported block type are compiled below into a mapping and thence compounded into a single function.

6.7.3 Block Virtualization Auxiliaries

The general policy in specification and implementation of virtualizers is to undertake optimization on-the-fly. This means that instead of constructing expressions for the output ports of virtualized block in a manner independent of the expressions for the input port values and then simplifying the result, the construction is undertaken in a manner sensitive to the expressions for the relevant input ports, and is optimal *ab initio*. This of course makes the constructors more complex and makes it more important not to duplicate construction specification and implementation, which is also desirable to limit the cost of enhancements to the optimization.

In this section belongs material concerned with analysis and construction of expressions which is not specific to some particular Simulink block type.

The following values are trivial kinds of *LIB_BLOCK_FUN1* and *LIB_BLOCK_FUN2*, where no construction of expressions is involved.

First where virtualisation is inhibited:

z

$$\begin{array}{|l}
\quad \boldsymbol{lbvf1\_inhibit}: LIB\_BLOCK\_VIRT\_FUN1;\\
\quad \boldsymbol{lbvf2\_inhibit}: LIB\_BLOCK\_VIRT\_FUN2\\
\hline
\quad (\forall emap: PVALUE \twoheadrightarrow Z\_EXPR \bullet\\
\qquad lbvf1\_inhibit\ emap = (VInhibit, \{\}));\\
\\
\quad (\forall mvartypes: MVARTYPES; alb: A\_LIB\_BLOCK; mv\_ctxt: \mathbb{P}\ PVALUE \bullet\\
\qquad lbvf2\_inhibit\ (mvartypes, alb, mv\_ctxt) =\\
\qquad\quad (lbvf1\_inhibit, \{\}))
\end{array}$$

The following provide generic support for source blocks, i.e. blocks which have no input ports. They are parameterized by an output expression map and a set of mask variables (used in the translated parameters), and request expressions for no input ports.

z

$$\begin{array}{|l}
\boldsymbol{lbvf1\_source}: (PVALUE \twoheadrightarrow Z\_EXPR) \times \mathbb{F}\ PVALUE \rightarrow LIB\_BLOCK\_VIRT\_FUN1;\\
\boldsymbol{lbvf2\_source}: (PVALUE \twoheadrightarrow Z\_EXPR) \times \mathbb{F}\ PVALUE \rightarrow LIB\_BLOCK\_VIRT\_FUN2\\
\hline
\quad (\forall oemap, iemap: PVALUE \twoheadrightarrow Z\_EXPR; used\_maskvars: \mathbb{F}\ PVALUE \bullet\\
\qquad lbvf1\_source\ (oemap, used\_maskvars)\ iemap\\
\qquad = (Virtual\ oemap, used\_maskvars));\\
\\
\quad (\forall iemap, oemap: PVALUE \twoheadrightarrow Z\_EXPR; used\_maskvars: \mathbb{F}\ PVALUE;\\
\quad\ \ mvartypes: MVARTYPES; alb: A\_LIB\_BLOCK; mv\_ctxt: \mathbb{P}\ PVALUE \bullet\\
\qquad lbvf2\_source\ (oemap, used\_maskvars)\ (mvartypes, alb, mv\_ctxt)\\
\qquad = (lbvf1\_source\ (oemap, used\_maskvars), \{\}))
\end{array}$$

Where virtualisation is trivial (e.g. for sinks) we have a source (because the input ports are irrelevant) with no output ports. No *LIB_BLOCK_VIRT_FUN1* need be specified.

z

$$\begin{array}{|l}
\quad \boldsymbol{lbvf2\_triv}: LIB\_BLOCK\_VIRT\_FUN2\\
\hline
\quad lbvf2\_triv = lbvf2\_source\ (\{\},\{\})
\end{array}$$

The following function derives a set of input portnames to be returned by a *LIB_BLOCK_FUN2*. It cross checks the number of input ports on the *Ports* param against the domain of the input port details, and returns the empty set as a failure indication if they do not match. Not to be used for blocks which have trigger or enable ports, or which have no *Ports* parameter or which have non-consecutive numeric input ports.

z

$$\begin{array}{|l}
\boldsymbol{ipset\_from\_ports\_param}: (\mathbb{F}\ PARAM) \times PORT\_INFO \rightarrow \mathbb{F}\ PVALUE \\
\hline
\forall pars: \mathbb{F}\ PARAM;\ pi: PORT\_INFO \bullet \\
\quad ipset\_from\_ports\_param\ (pars,\ pi) = \\
\\
\quad \{\ pn,\ ppv: PVALUE;\ pp: PORT\_PARAM \\
\quad \mid (name \mathrel{\widehat=} Ports,\ value \mathrel{\widehat=} ppv) \in pars \\
\quad \land\ ppv \mapsto pp \in parse\_ports\_param \\
\quad \land\ dom\ pi.input\_port\_details = num2pvalue\ (\!|1..pp.max\_in|\!) \\
\quad \land\ pn \in dom\ pi.input\_port\_details \\
\quad \bullet\ pn \\
\quad \}
\end{array}$$

Next we have auxiliaries which are used for translation of parameter expressions (e.g. for constant blocks).

The parameters to this function are:

- a set of *PARAM*s
- the set of free mask variables

then a map assigning:

- the name of a parameter to be translated
- a parameter translation code

to output port names (as *PVALUE*s).

The result is:

- a map assigning to output port names the *Z_EXPR* of the translated parameter. As specified, if translation fails the relevant portname simply has no value assigned to it in the resulting map.
- the set of mask variables used in the translations

This function is provided supporting a set of parameter translations since this is the easiest way to deal with the possibility that there will be *no* sucessful translations, so far we have no blocks with more than one expression to translate.

z

$$\begin{array}{l}
\boldsymbol{param\_zexp}: (\mathbb{F}\ PARAM) \times \mathbb{F}\ PVALUE \\
\quad \to (PVALUE \twoheadrightarrow PNAME \times PVALUE) \\
\quad \to (PVALUE \twoheadrightarrow Z\_EXPR) \times \mathbb{F}\ PVALUE \\
\hline
\forall\ params: \mathbb{F}\ PARAM;\ mv\_ctxt: \mathbb{F}\ PVALUE;\ pmap: PVALUE \twoheadrightarrow PNAME \times PVALUE \bullet \\
param\_zexp\ (params,\ mv\_ctxt)\ pmap = \\
(\mu amp{:}\{\{op,\ trc,\ pv: PVALUE;\ pn: PNAME;\ ze: Z\_EXPR;\ used\_maskvars: \mathbb{F}\ PVALUE \\
\quad \mid op \mapsto (pn,\ trc) \in pmap \\
\quad \land (name \mathrel{\widehat{=}} pn,\ value \mathrel{\widehat{=}} pv) \in params \\
\quad \land (TMatch\ ze,\ used\_maskvars) = par\_trans2\ mv\_ctxt\ trc\ pv \\
\quad \bullet (op \mapsto ze,\ used\_maskvars)\}\} \\
\ \bullet (first(\!|amp|\!),\ \bigcup\ (second(\!|amp|\!))))
\end{array}$$

The following auxiliary is for parameters which must be translated by *special_par_trans*, e.g. the selector block. The structure is broadly similar, but *SPECIAL_RESULT*s and *PARAMETER*s are returned rather than *Z_EXPR*s.

z

$$\begin{array}{l}
\boldsymbol{param\_sr}: (\mathbb{F}\ PARAM) \times (\mathbb{F}\ PVALUE) \times MVARTYPES \\
\quad \to (PVALUE \twoheadrightarrow PNAME \times PORT\_TYPE) \\
\quad \to (PVALUE \twoheadrightarrow PARAMETER \times SPECIAL\_RESULT) \times \mathbb{F}\ PVALUE \\
\hline
\forall\ params: \mathbb{F}\ PARAM;\ mv\_ctxt: \mathbb{F}\ PVALUE;\ mvartypes: MVARTYPES; \\
\ \ pmap: PVALUE \twoheadrightarrow PNAME \times PORT\_TYPE \bullet \\
param\_sr\ (params,\ mv\_ctxt,\ mvartypes)\ pmap = \\
(\mu amp{:}\{\{op,\ pv: PVALUE;\ pt: PORT\_TYPE;\ pn: PNAME;\ ze: Z\_EXPR; \\
\qquad parameter: PARAMETER;\ sr: SPECIAL\_RESULT;\ used\_maskvars: \mathbb{F}\ PVALUE \\
\quad \mid op \mapsto (pn,\ pt) \in pmap \\
\quad \land (name \mathrel{\widehat{=}} pn,\ value \mathrel{\widehat{=}} pv) \in params \\
\quad \land (parameter,\ sr,\ used\_maskvars) \\
\qquad = special\_par\_trans\_wp\ mvartypes\ mv\_ctxt\ pt\ pv \\
\quad \bullet (op \mapsto (parameter,\ sr),\ used\_maskvars)\}\} \\
\ \bullet (first(\!|amp|\!),\ \bigcup\ (second(\!|amp|\!))))
\end{array}$$

We now present specifications which are concerned with the analysis of expressions (for signal values) on which constructions are to be performed, and with constructions optimised on the basis of the analyses. It is intended that these be presented in as generic a way as is feasible, both with respect to their possible application to multiple Simulink block types, and with respect to their application in synthesis as well as in virtualization. The synthesis method does not permit optimisation, but the constructions in the cases where no optimisation is possible correspond to the construction of expressions for the predicates of synthesized block schemas.

It is not intended that existing synthesis functions will be rewritten to use the specifications here, but that support for synthesis of additional block types will do so when appropriate.

The initial offering on optimisation is basic, and the analysis necessary to support that is elementary. The analysis is packaged and used in a manner which is intended to facilitate extension of the analysis to support a wider range of optimisations.

The initial analysis distinguishes only three kinds of signal expression.

- vector displays (constructor *ZSequence*)
- bus contructions (constructor *ZBus*)
- other expressions

The following operators are provided to facilitate construction of specifications which are conditional on the kind of input signal.

The first is a three way switch:

z

$$
\begin{array}{l}
[X] \\
\hline
\boldsymbol{signal\_kind\_switch}: \\
\quad (seq\ Z\_EXPR \nrightarrow X) \times (seq\ Z\_EXPR \nrightarrow X) \times (Z\_EXPR \nrightarrow X) \\
\rightarrow (Z\_EXPR \nrightarrow X) \\
\hline
\forall\ ze: Z\_EXPR;\ vecpf,\ buspf: seq\ Z\_EXPR \nrightarrow X;\ opf: Z\_EXPR \nrightarrow X \bullet \\
signal\_kind\_switch\ (vecpf,\ buspf,\ opf) \\
= \{ze: Z\_EXPR;\ sze: seq\ Z\_EXPR;\ x{:}X \\
\quad \mid ze = ZSequence\ sze \\
\quad\quad \land sze \mapsto x \in vecpf \\
\quad\ \lor ze = ZBus\ sze \\
\quad\quad \land sze \mapsto x \in buspf \\
\quad \bullet ze \mapsto x\} \\
\cup \{ze: Z\_EXPR;\ x{:}X \\
\quad \mid \neg(\exists sze: seq\ Z\_EXPR \bullet ze = ZSequence\ sze \lor ze = ZBus\ sze) \\
\quad \land ze \mapsto x \in opf \\
\quad \bullet ze \mapsto x\}
\end{array}
$$

A couple of two way switches are also defined. The first for treating displays only as a special case.

z

$$
\begin{array}{l}
[X] \\
\quad \boldsymbol{display\_switch}: \\
\qquad (seq\ Z\_EXPR \nrightarrow X) \times (Z\_EXPR \nrightarrow X) \\
\quad \rightarrow (Z\_EXPR \nrightarrow X) \\
\hline
\quad \forall\ ze: Z\_EXPR;\ vecpf: seq\ Z\_EXPR \nrightarrow X;\ opf: Z\_EXPR \nrightarrow X \bullet \\
\quad display\_switch\ (vecpf,\ opf) \\
\quad = signal\_kind\_switch \\
\qquad\qquad (vecpf,\ opf\ \ o\ ZBus,\ opf)
\end{array}
$$

The second for treating *ZBus* only as a special case.

z

$$
\begin{array}{l}
[X] \\
\quad \boldsymbol{bus\_switch}: \\
\qquad (seq\ Z\_EXPR \nrightarrow X) \times (Z\_EXPR \nrightarrow X) \\
\quad \rightarrow (Z\_EXPR \nrightarrow X) \\
\hline
\quad \forall\ ze: Z\_EXPR;\ buspf: seq\ Z\_EXPR \nrightarrow X;\ opf: Z\_EXPR \nrightarrow X \bullet \\
\quad bus\_switch\ (buspf,\ opf) \\
\quad = signal\_kind\_switch \\
\qquad\qquad (opf\ \ o\ ZSequence,\quad buspf,\ opf)
\end{array}
$$

When a construction takes place over a set of inputs (e.g. bus construction or mux) it is desirable to know whether the set of inputs can be combined into a single sequence display. This depends not only on the *SIGNAL_KIND* but also on the *PORT_TYPE*, the condition being that all but scalar inputs are themselves sequence displays.

This separation of testing for this condition from the composition of a single vector display is probably not useful so this function does the construction if it can be done. Care should be taken in implementation to get the domain of this function correct.

z

$$
\begin{array}{|l}
\boldsymbol{display\_from\_inputs}: (PVALUE \twoheadrightarrow PORT\_DETAILS) \times (PVALUE \twoheadrightarrow Z\_EXPR) \\
\qquad\qquad\qquad \rightarrowtail Z\_EXPR \\
\hline
\forall\ pdmap: PVALUE \twoheadrightarrow PORT\_DETAILS;\ zemap: PVALUE \twoheadrightarrow Z\_EXPR;\ ze: Z\_EXPR \bullet \\
\quad (pdmap, zemap) \mapsto ze \in display\_from\_inputs \\
\quad \Leftrightarrow \\
\quad (\exists\ ssze: seq\ seq\ Z\_EXPR \\
\quad \mid dom\ pdmap = dom\ zemap = num2pvalue\ (\!|\ 1..(\#ssze)\ |\!) \\
\quad \wedge (\forall\ ip: dom\ pdmap;\ n: dom\ ssze \mid ip = num2pvalue\ n \bullet \\
\qquad\quad ((pdmap\ ip).port\_type = ScalarPT \\
\qquad\quad \Rightarrow ssze\ n = \langle zemap\ ip \rangle) \\
\qquad \wedge \quad ((pdmap\ ip).port\_type \neq ScalarPT \\
\qquad\quad \Rightarrow ZSequence\ (ssze\ n) = zemap\ ip)) \\
\quad \bullet\ ze = ZSequence\ (\frown/\ ssze))
\end{array}
$$

Where a sequence display is not possible the results are to be combined using *ZBus* so that the bus structure remains transparent. Again it is necessary to know the port types.

z

$$
\begin{array}{|l}
\boldsymbol{zbus\_from\_inputs}: (PVALUE \twoheadrightarrow PORT\_DETAILS) \times (PVALUE \twoheadrightarrow Z\_EXPR) \\
\qquad\qquad\qquad \rightarrowtail Z\_EXPR \\
\hline
\forall\ pdmap: PVALUE \twoheadrightarrow PORT\_DETAILS;\ zemap: PVALUE \twoheadrightarrow Z\_EXPR;\ ze: Z\_EXPR \bullet \\
\quad (pdmap, zemap) \mapsto ze \in zbus\_from\_inputs \\
\quad \Leftrightarrow \\
\quad (\exists\ seqze: seq\ Z\_EXPR \\
\quad \mid dom\ pdmap = dom\ zemap = num2pvalue\ (\!|\ 1..(\#seqze)\ |\!) \\
\quad \wedge (\forall\ ip: dom\ pdmap;\ n: dom\ seqze;\ pt: PORT\_TYPE \\
\qquad \mid ip = num2pvalue\ n \\
\qquad \wedge\ pt = (pdmap\ ip).port\_type \\
\qquad \bullet\ pt \neq UnknownPT \\
\qquad \wedge\ (pt = ScalarPT \Rightarrow seqze\ n = ZSequence\ \langle zemap\ ip \rangle) \\
\qquad \wedge\ (pt \neq ScalarPT \Rightarrow seqze\ n = zemap\ ip)) \\
\quad \bullet\ ze = ZBus\ seqze)
\end{array}
$$

6.7.4 Virtualize Constant

The translated *Value* parameter is used as the output expression for port *One*. A possible enhancement would be to take account of the type on the output port, as is done for the *Selector* block.

z

$$
\begin{array}{l}
\boldsymbol{virtualize\_constant}: LIB\_BLOCK\_VIRT\_FUN2 \\
\hline
\forall\ mvartypes: MVARTYPES;\ alb: A\_LIB\_BLOCK;\ mv\_ctxt: \mathbb{F}\ PVALUE \bullet \\
virtualize\_constant\ (mvartypes,\ alb,\ mv\_ctxt) \\
=\ lbvf2\_source \\
\quad (param\_zexp\ (alb.block\_info.pars,\ mv\_ctxt)\ \{One \mapsto (ValuePN,\ SVM)\}) \\
\quad (mvartypes,\ alb,\ mv\_ctxt)
\end{array}
$$

6.7.5 Virtualize Bus Selector

Different methods of selection are appropriate depending on whether the input is a *ZBus* or not. This diffence applies at each stage in a multistage selection, so there could be several stages of selection from *ZBus* constructors, possibly followed by selections either from a vector display or from some other kind of expression. First we specify the expressions for a single output, then if muxed output is required these will be combined using *ZBus*.

The following function determines the location of a selection from a bus, as a pair of natural numbers. These values are required when the bus is not presented as a *ZBus*, and also during synthesis.

z

$$
\begin{array}{l}
\boldsymbol{selection\_position}: PORT\_TYPE \rightarrow seq\ PVALUE \mathrel{+\!\!\!\rightarrow} (\mathbb{N} \times \mathbb{N}) \\
\hline
\forall\ pt2: PORT\_TYPE;\ spv: seq\ PVALUE;\ lp,\ rp: \mathbb{N} \bullet \\
\quad spv \mapsto (lp,\ rp) \in selection\_position\ pt2 \\
\Leftrightarrow \\
\quad (\exists\ w: \mathbb{N};\ bdets: seq\ PORT\_DETAILS \bullet \\
\quad pt2 = BusPT\ (w,\ bdets) \\
\quad \land\ (\quad spv = \langle\rangle \land lp = 1 \land rp = w \\
\quad\quad \lor\quad (\exists\ p,\ rlp,\ rrp,\ offset: \mathbb{N};\ h: PVALUE;\ t: seq\ PVALUE \\
\quad\quad\quad \mid \langle h\rangle \frown t = spv \\
\quad\quad\quad \land ((bdets\ p).line\_name = h \\
\quad\quad\quad\quad \lor \neg(\exists pd{:}ran\ bdets \bullet\ pd.line\_name = h) \\
\quad\quad\quad\quad\quad \land\ h = port2signal\ (num2pvalue\ p)) \\
\quad\quad\quad \land\ t \mapsto (rlp,\ rrp) \in selection\_position\ (bdets\ p).port\_type \\
\quad\quad\quad \land\ offset = nat\_seq\_sum\ ((id(1..(p-1)))\ ⨾\ bdets\ ⨾\ pd\_width) \\
\quad\quad\quad \bullet\ lp = rlp + offset \\
\quad\quad\quad \land\ rp = rrp + offset) \\
\quad\quad) \\
\quad) \\
\lor\quad (\exists\ w: \mathbb{N} \bullet\ pt2 = VectorPT\ w \land spv = \langle\rangle \land lp = 1 \land rp = w) \\
\lor\quad (pt2 = ScalarPT \land spv = \langle\rangle \land lp = 1 \land rp = 1)
\end{array}
$$

The following defines the construction of a selection expression which delivers a vector:

```
z
│    vector_selection_exp: (ℕ × ℕ) → Z_EXPR ⇸ Z_EXPR
├──────────────────────────────
│    ∀ lp, rp: ℕ •
│    vector_selection_exp (lp, rp)
│    = display_switch (
│    (λsze: seq Z_EXPR• ZSequence ((lp :z rp) ⨾ sze)),
│    (λze: Z_EXPR• ZInfixOps
│             (ze,
│              ⟨(Composei,
│                ZBrackets (
│                  ZInfixOps
│                      (PvalueZexpr (num2pvalue lp),
│                       ⟨(pvalue2ident (sc2pv ":z"),
│                         PvalueZexpr (num2pvalue rp)
│                       )⟩
│                      )
│                )
│              )⟩
│             )
│    ))
```

A scalar selection is done by function application.

```
z
│    scalar_selection_exp: ℕ → Z_EXPR ⇸ Z_EXPR
├──────────────────────────────
│ ∀ p: ℕ•
│ scalar_selection_exp p
│ = display_switch (
│         (λsze: seq Z_EXPR | p ∈ dom sze • sze p),
│         (λze: Z_EXPR• Application
│                  (ZBrackets ze,
│                  ZBrackets(PvalueZexpr (num2pvalue p)))))
```

Now we combine these using the width as selection criterion.

z

$$\boldsymbol{selection\_exp}: (\mathbb{N} \times \mathbb{N}) \rightarrow Z\_EXPR \pinj Z\_EXPR$$

$$\begin{array}{l}
\forall\ ize,\ oze: Z\_EXPR;\ lp,\ rp: \mathbb{N} \bullet \\
ize \mapsto oze \in selection\_exp\ (lp,\ rp) \\
\Leftrightarrow if\ \ lp = rp \\
\quad\quad then\ ize \mapsto oze \in scalar\_selection\_exp\ lp \\
\quad\quad else\ (ize \mapsto oze \in vector\_selection\_exp\ (lp,\ rp))
\end{array}$$

The following specification concerns bus selection on a signal whose *Z_EXPR* is *not* a *ZBus*. This may be selection using the tail end of a selection on a component of a *ZBus*. The procedure is first to calculate (using the port type and the selection information) the position of the required portion of the signal (as first and last indexes), and then to make the selection. The way in which the selection is made depends upon whether the expression is a vector display. If it is, the required slice of the display is computed and returned as a vector display, otherwise the expression is composed with the relevant interval. Note that the port type supplied to *nonbus_bus_selection* need not be a *BusPT*. Like *selection_position* it is specified to cover other types as the base cases of a recursion down the signal structure. It should not be used for bus selection from *Z_EXPR* which is a *ZBus* since it will deliver a suboptimal expression in that case.

z

$$\begin{array}{l}
\boldsymbol{nonbus\_bus\_selection}: (PORT\_TYPE \times seq\ PVALUE) \\
\quad\quad \rightarrow (Z\_EXPR \pinj Z\_EXPR \times \mathbb{N})
\end{array}$$

$$\begin{array}{l}
\forall\ ize,\ oze: Z\_EXPR;\ pt: PORT\_TYPE;\ spv: seq\ PVALUE;\ w: \mathbb{N} \bullet \\
ize \mapsto (oze,\ w) \in nonbus\_bus\_selection\ (pt,\ spv) \\
\Leftrightarrow \\
(\exists\ lp,\ rp: \mathbb{N} \\
\mid\ \ spv \mapsto (lp,\ rp) \in selection\_position\ \ pt \\
\bullet\ \ ize \mapsto oze \in selection\_exp\ (lp,\ rp) \\
\land\ \ w = rp + 1 - lp \\
)
\end{array}$$

This function is for the case that the *Z_EXPR* for the bus is a *ZBus*. It expects the *ZBus* to have been stripped off, hence requiring a *seq Z_EXPR* parameter.

z

$$
\begin{array}{|l}
\boldsymbol{bus\_bus\_selection}: \\
\quad (seq\ Z\_EXPR) \times (seq\ PORT\_DETAILS) \times PVALUE \\
\qquad \nrightarrow (Z\_EXPR \times PORT\_TYPE) \\
\hline
\qquad \forall\ spd: seq\ [line\_name: PVALUE; port\_type: PORT\_TYPE]; \\
\qquad\quad pv: PVALUE; sze: seq\ Z\_EXPR; oze: Z\_EXPR; opt: PORT\_TYPE \bullet \\
\qquad (sze, spd, pv) \mapsto (oze, opt) \in bus\_bus\_selection \\
\qquad \Leftrightarrow \\
\qquad (\exists\ n: \mathbb{N}; sze: seq\ Z\_EXPR \\
\qquad \mid\ oze = sze\ n \\
\qquad \land\ opt = (spd\ n).port\_type \\
\qquad \bullet\ ((spd\ n).line\_name = pv \\
\qquad\quad \lor \neg(\exists pd: ran\ spd \bullet pd.line\_name = pv) \\
\qquad\qquad \land\ pv = port2signal\ (num2pvalue\ n) \\
\qquad\quad) \\
\qquad)
\end{array}
$$

The following specification provides a map of expressions for the outputs of a bus selector together with the width of each output. Outputs of unit width are scalars, the rest are vectors. The outputs may later be muxed together, and in that case the output width is used to determine which are scalars (and therefore need to be made into unit vectors for muxing).

z

$$
\begin{array}{|l}
\boldsymbol{bus\_selector\_outputw}: Z\_EXPR \times PORT\_TYPE \rightarrow (seq\ PVALUE \nrightarrow Z\_EXPR \times \mathbb{N}) \\
\hline
(\forall\ ize,\ oze: Z\_EXPR;\ pt: PORT\_TYPE;\ w: \mathbb{N} \bullet \\
\langle\rangle \mapsto (oze,\ w) \in bus\_selector\_outputw\ (ize,\ pt) \\
\Leftrightarrow oze = ize \land w = pt2\_width\ pt); \\
\\
(\forall\ ize,\ oze: Z\_EXPR;\ pt: PORT\_TYPE;\ w: \mathbb{N};\ hpv: PVALUE;\ spv: seq\ PVALUE \bullet \\
(\langle hpv\rangle \frown spv) \mapsto (oze,\ w) \in bus\_selector\_outputw\ (ize,\ pt) \\
\Leftrightarrow \\
ize \mapsto (oze,\ w) \in bus\_switch\ (\\
\quad\quad (\mu\ n: \mathbb{N};\ ssd: seq\ [line\_name: PVALUE;\ port\_type: PORT\_TYPE] \\
\quad\quad \mid pt = BusPT\ (n,\ ssd) \\
\quad\quad \bullet \{sze: seq\ Z\_EXPR;\ ze1,\ ze2: Z\_EXPR;\ pt2: PORT\_TYPE;\ w2: \mathbb{N} \\
\quad\quad\ \ \mid (sze,\ ssd,\ hpv) \mapsto (ze1,\ pt2) \in bus\_bus\_selection \\
\quad\quad\ \ \land spv \mapsto (ze2,\ w2) \in bus\_selector\_outputw\ (ze1,\ pt2) \\
\quad\quad\ \ \bullet sze \mapsto (ze2,\ w2)\}), \\
\quad\quad nonbus\_bus\_selection\ (pt,\ \langle hpv\rangle \frown spv) \\
\quad\quad) \\
)
\end{array}
$$

The further transformation of this information depends on whether "MuxedOutput" is selected.

If the selection is to be used for a single output line then the type of the output line and the mode of selection will both depend upon the width of the signal, a signal of width 1 is delivered as a scalar.

z

$$
\begin{array}{|l}
\quad\quad \boldsymbol{nonmux\_selection}: (Z\_EXPR \times PORT\_TYPE) \\
\quad\quad\quad\quad \rightarrow (seq\ PVALUE \nrightarrow Z\_EXPR) \\
\hline
\quad\quad \forall\ ze,\ ze': Z\_EXPR;\ pt2: PORT\_TYPE;\ spv: seq\ PVALUE \bullet \\
\quad\quad spv \mapsto ze' \in nonmux\_selection\ (ze,\ pt2) \\
\quad\quad \Leftrightarrow \\
\quad\quad (\exists w: \mathbb{N} \bullet spv \mapsto (ze',\ w) \in bus\_selector\_outputw\ (ze,\ pt2))
\end{array}
$$

If the signal is required for use in a muxed output then it will have to be made into a vector so the the various outputs can be combined by *ZBus*.

z

$$\begin{array}{l}
\boldsymbol{mux\_selection}: (Z\_EXPR \times PORT\_TYPE) \\
\quad \rightarrow (seq\ PVALUE \pfun Z\_EXPR) \\
\hline
\forall ze, ze': Z\_EXPR; pt2: PORT\_TYPE; spv: seq\ PVALUE \bullet \\
spv \mapsto ze' \in mux\_selection\ (ze, pt2) \\
\Leftrightarrow \\
(\exists w: \mathbb{N} \bullet spv \mapsto (ze', w) \in bus\_selector\_outputw\ (ze, pt2) \\
\quad \land w > 1) \\
\lor \\
(\exists scze: Z\_EXPR \bullet spv \mapsto (scze, 1) \in bus\_selector\_outputw\ (ze, pt2) \\
\quad \land ze' = ZSequence\ \langle scze \rangle)
\end{array}$$

The following function converts a port details map (which might contain non-numeric ports like *Trigger*) to a sequence containing just the details for the numeric ports. Note that this function fails to deliver a result if there are no (positive) numeric output ports or if there are gaps in the sequence of numeric ouput ports. It is now generic, applying to any map whose domain is a set of *PVALUE*s which is an initial segment of the positive natural numbers.

z

$$\begin{array}{l}
[X] \\
\hline
\boldsymbol{numeric\_port\_sequence}: (PVALUE \ffun X) \pfun seq\ X \\
\hline
\forall pds: PVALUE \ffun X; spds: seq\ X \bullet \\
\quad pds \mapsto spds \in numeric\_port\_sequence \\
\Leftrightarrow \\
\quad spds = num2pvalue \fatsemi pds
\end{array}$$

We now provide a specification of how to obtain the correct ouput expression map by referring to the *MuxedOutput* parameter and then using the appropriate choice of the above two methods.

z

$$\begin{array}{l}
\boldsymbol{bus\_selector\_outputs}: (\mathbb{F}\ PARAM) \times PORT\_TYPE \\
\quad \rightarrow (Z\_EXPR \pfun (PVALUE \ffun Z\_EXPR)) \\
\hline
\forall\ pars: \mathbb{F}\ PARAM;\ pt: PORT\_TYPE;\ ze: Z\_EXPR;\ zemap: PVALUE \pfun Z\_EXPR \bullet \\
ze \mapsto zemap \in bus\_selector\_outputs\ (pars,\ pt) \\
\Leftrightarrow \\
(\exists\ par: pars;\ osp: OUTPUTSIGNALS\_PARAM;\ mopar: PARAM;\ sze: seq\ Z\_EXPR \\
\bullet\ par.name = OutputSignals \\
\land\ par.value \mapsto osp \in parse\_outputsignals\_param \\
\land\ mopar = (name \mathrel{\widehat=} sc2pn\ \texttt{"}MuxedOutput\texttt{"},\ value \mathrel{\widehat=} on) \\
\land\ sze = osp \fatsemi (\\
\quad (if\ \ mopar \in pars\ \ then\ \ mux\_selection\ \ else\ \ nonmux\_selection) \\
\quad (ze,\ pt)) \\
\land\ zemap = \quad if\ \ mopar \in pars \\
\qquad\qquad then\ \{One \mapsto ZBus\ \ sze\} \\
\qquad\qquad else\ \ (numeric\_port\_sequence\ ^{\sim})\ \ sze)
\end{array}$$

Finally the *LIB_BLOCK_VIRT_FUN2* for *BusSelector*:

z
```
    virtualize_bus_selector: LIB_BLOCK_VIRT_FUN2
 ─────────────────────────────────────
 ∀ mvartypes: MVARTYPES; alb: A_LIB_BLOCK; mv_ctxt: 𝔽 PVALUE•
    virtualize_bus_selector (mvartypes, alb, mv_ctxt) =
    (μ lbvf1: LIB_BLOCK_VIRT_FUN1
    | lbvf1 =
        if dom alb.port_info.input_port_details ≠ {One}
        then lbvf1_inhibit
        else
     {zemap: PVALUE ⇸ Z_EXPR; v: VIRTUAL
     | v =if dom zemap = {One}
        then if (zemap One) ∈ dom (bus_selector_outputs (
                    alb.block_info.pars,
                    (alb.port_info.input_port_details One).port_type))
            then Virtual (bus_selector_outputs (alb.block_info.pars,
                        (alb.port_info.input_port_details One).port_type)
                        (zemap One))
            else VInhibit
        else VInhibit
     • zemap ↦ (v, {})}
    • (lbvf1, {One})
    )
```

6.7.6 Virtualize Mux

For *Mux* (and *BusConstructor*) if possible (i.e. if all the inputs are vector displays or scalar expressions) a vector display is used for the output, failing that the *ZBus* constructor is used to combine the input signals.

```
z
│     virtualize_mux: LIB_BLOCK_VIRT_FUN2
├──────────────────────────────
│ ∀ mvartypes: MVARTYPES; alb: A_LIB_BLOCK; mv_ctxt: 𝔽 PVALUE•
│     virtualize_mux (mvartypes, alb, mv_ctxt) =
│     (μ lbvf1: LIB_BLOCK_VIRT_FUN1; inports: 𝔽 PVALUE
│     | inports = ipset_from_ports_param (alb.block_info.pars, alb.port_info)
│     ∧ lbvf1 =
│             if inports = {}
│             then lbvf1_inhibit
│             else
│       {zemap: PVALUE ⇸ Z_EXPR; v: VIRTUAL
│       | v = if dom zemap = dom alb.port_info.input_port_details
│             then if (alb.port_info.input_port_details, zemap)
│                     ∈ dom display_from_inputs
│                 then Virtual {One ↦
│                     display_from_inputs (alb.port_info.input_port_details, zemap)}
│                 else if (alb.port_info.input_port_details, zemap)
│                     ∈ dom zbus_from_inputs
│                 then Virtual {One ↦
│                     zbus_from_inputs (alb.port_info.input_port_details, zemap)}
│                 else VInhibit
│             else VInhibit
│       • zemap ↦ (v, {})}
│     • (lbvf1, inports)
│     )
```

6.7.7 Virtualize Demux

The following functions are used in deciding which slice of an input vector or bus to output along the various output lines of a *Demux* block. This is done using the output port details of the *Demux*, ignoring the input port structure (which will have been used to establish the output port details earlier).

z

$$\begin{array}{|l}
\quad \boldsymbol{demux\_selection\_pos}: seq\ PORT\_DETAILS \nrightarrow (seq\ \mathbb{N} \times \mathbb{N}) \times \mathbb{N} \\
\hline
\quad \langle\rangle \mapsto (\langle\rangle,\ 0) \in demux\_selection\_pos \\
\land \\
\quad (\forall\ spds: seq\ PORT\_DETAILS;\ pd: PORT\_DETAILS;\ spos: seq\ (\mathbb{N} \times \mathbb{N});\ w: \mathbb{N} \bullet \\
\quad spds \frown \langle pd\rangle \mapsto (spos,\ w) \in demux\_selection\_pos \\
\quad \Leftrightarrow \\
\quad (\exists\ spos2: seq\ (\mathbb{N} \times \mathbb{N});\ wf,\ wl: \mathbb{N} \\
\quad \mid spds \mapsto (spos2,\ wf) \in demux\_selection\_pos \\
\quad \land\ wl = pd\_width\ pd \land wl > 0 \\
\quad \bullet\ spos = spos2 \frown \langle(wf{+}1,\ wf{+}wl)\rangle \\
\quad \land\ w = wf + wl))
\end{array}$$

Now we specify an output port expression sequence.

z

$$\begin{array}{|l}
\quad \boldsymbol{demux\_exp\_seq}: (seq\ PORT\_DETAILS) \rightarrow Z\_EXPR \nrightarrow (seq\ Z\_EXPR) \\
\hline
\forall\ pdseq: seq\ PORT\_DETAILS;\ ize: Z\_EXPR;\ ozeseq: seq\ Z\_EXPR \bullet \\
\quad ize \mapsto ozeseq \in demux\_exp\_seq\ pdseq \\
\Leftrightarrow \\
\quad (\exists\ spos: seq\ (\mathbb{N} \times \mathbb{N});\ w: \mathbb{N} \\
\quad\ \mid pdseq \mapsto (spos,\ w) \in demux\_selection\_pos \\
\quad\ \bullet \forall\ n: \mathbb{N};\ oze: Z\_EXPR \\
\quad\quad \bullet\ n \mapsto oze \in ozeseq \Leftrightarrow \\
\quad\quad\quad\quad n \in dom\ pdseq \\
\quad\quad\quad \land\ ize \mapsto oze \in selection\_exp\ (spos\ n))
\end{array}$$

Now we specify an output port expression map.

z

$$\begin{array}{|l}
\quad \boldsymbol{demux\_exp\_map}: (PVALUE \twoheadrightarrow PORT\_DETAILS) \\
\quad\quad\quad\quad\quad \rightarrow Z\_EXPR \nrightarrow (PVALUE \twoheadrightarrow Z\_EXPR) \\
\hline
\forall\ pdmap: PVALUE \twoheadrightarrow PORT\_DETAILS;\ ize: Z\_EXPR; \\
\quad ozemap: PVALUE \twoheadrightarrow Z\_EXPR \bullet \\
\quad ize \mapsto ozemap \in demux\_exp\_map\ pdmap \\
\Leftrightarrow \\
\quad (\exists\ pdseq: seq\ PORT\_DETAILS;\ ozeseq: seq\ Z\_EXPR \\
\quad \mid pdmap \mapsto pdseq \in numeric\_port\_sequence \\
\quad \land\ ozemap \mapsto ozeseq \in numeric\_port\_sequence \\
\quad \bullet\ ize \mapsto ozeseq \in demux\_exp\_seq\ pdseq)
\end{array}$$

And finally *Demux* virtualization:

```
z
│    virtualize_demux: LIB_BLOCK_VIRT_FUN2
├──────────────────────────────────────
│ ∀ mvartypes: MVARTYPES; alb: A_LIB_BLOCK; mv_ctxt: 𝔽 PVALUE•
│    virtualize_demux (mvartypes, alb, mv_ctxt) =
│    (μ lbvf1: LIB_BLOCK_VIRT_FUN1; inports: 𝔽 PVALUE
│    | inports = ipset_from_ports_param (alb.block_info.pars, alb.port_info)
│    ∧ lbvf1 =
│            if inports ≠ {One}
│            then lbvf1_inhibit
│            else
│      {zemap: PVALUE ⇸ Z_EXPR; v: VIRTUAL
│      | v = if dom zemap = {One}
│            then if (zemap One) ∈ dom (demux_exp_map
│                            alb.port_info.output_port_details)
│                then Virtual
│                            (demux_exp_map
│                            alb.port_info.output_port_details
│                            (zemap One))
│                else VInhibit
│            else VInhibit
│      • zemap ↦ (v, {})}
│    • (lbvf1, {One})
│    )
```

6.7.8 Virtualize Bus Creator

BusCreator is treated exactly the same as *Mux*.

```
z
│    virtualize_bus_creator: LIB_BLOCK_VIRT_FUN2
├──────────────────────────────────────
│    virtualize_bus_creator = virtualize_mux
```

6.7.9 Virtualize Selector

For simplification of selector virtualizations we need to be able to select from a vector display.

z

$$
\begin{array}{|l}
\boldsymbol{select\_from\_seq\_by\_real}: (seq\ Z\_EXPR) \rightarrow (seq\ \mathbb{R}) \rightarrowtail\!\!\!\!\!\rightarrow (seq\ Z\_EXPR) \\
\hline
\forall\ iseqze, oseqze: seq\ Z\_EXPR; seqr: seq\ \mathbb{R} \bullet \\
\qquad seqr \mapsto oseqze \in select\_from\_seq\_by\_real\ iseqze \\
\Leftrightarrow \\
\qquad (\exists\ seqn: seq\ \mathbb{N} \\
\qquad \mid seqn \mathbin{⨾} real = seqr \\
\qquad \bullet\ oseqze = seqn \mathbin{⨾} iseqze)
\end{array}
$$

This could be generic, but its doubtful that any other uses would arise. Given the sequence of expressions in some vector display this function specifies the subsequence selected by a *PARAMETER* which is expected to be a parsed *Elements* parameter.

z

$$
\begin{array}{|l}
\boldsymbol{select\_from\_seq\_by\_param}: (seq\ Z\_EXPR) \rightarrow PARAMETER \rightarrowtail\!\!\!\!\!\rightarrow (seq\ Z\_EXPR) \\
\hline
\forall\ iseqze, oseqze: seq\ Z\_EXPR; parameter: PARAMETER \bullet \\
\qquad parameter \mapsto oseqze \in select\_from\_seq\_by\_param\ iseqze \\
\Leftrightarrow \\
\qquad (\exists\ vexp: VEXP; seqr: seq\ \mathbb{R} \\
\qquad \mid ParVector\ vexp = parameter \\
\qquad \land\ vexp \mapsto seqr \in vexp\_val \\
\qquad \bullet\ seqr \mapsto oseqze \in select\_from\_seq\_by\_real\ iseqze) \\
\qquad \lor \\
\qquad (\exists\ sexp: SEXP; n: \mathbb{N}; ze: Z\_EXPR \\
\qquad \mid ParScalar\ sexp = parameter \\
\qquad \land\ sexp \mapsto (real\ n) \in sexp\_val \\
\qquad \land\ n \mapsto ze \in iseqze \\
\qquad \bullet\ oseqze = \langle ze \rangle)
\end{array}
$$

The following function computes the expression for a *Selector* block when the input is *not* a sequence display. There are opportunities for further optimization here.

z

$$
\begin{array}{l}
\boldsymbol{selector\_expression\_from\_other}: SPECIAL\_RESULT \rightarrow (Z\_EXPR \nrightarrow Z\_EXPR) \\
\hline
\forall\ sr: SPECIAL\_RESULT;\ ize: Z\_EXPR;\ oze: Z\_EXPR \bullet \\
\quad ize \mapsto oze \in selector\_expression\_from\_other\ sr \\
\Leftrightarrow \quad (\exists\ ze: Z\_EXPR \\
\quad \bullet\ sr = SRVector\ ze \\
\qquad \land\ oze = ZInfixOps\ (ize, \\
\qquad\qquad \langle (Composei,\ Ident\ R2zi), \\
\qquad\qquad (Composei,\ ZBrackets\ ze)\rangle) \\
\quad \lor\ sr = SRScalar\ ze \\
\qquad \land\ oze = Application\ (ize,\ ZBrackets\ (Application\ (Ident\ R2zi,\ ze))))
\end{array}
$$

Now we specify for the sequence display case.

z

$$
\begin{array}{l}
\boldsymbol{selector\_expression\_from\_display}: PARAMETER \times SPECIAL\_RESULT \\
\qquad\qquad \rightarrow (seq\ Z\_EXPR \nrightarrow Z\_EXPR) \\
\hline
\forall\ iseqze: seq\ Z\_EXPR;\ p: PARAMETER;\ sr: SPECIAL\_RESULT;\ oze: Z\_EXPR \bullet \\
\quad iseqze \mapsto oze \in selector\_expression\_from\_display\ (p,\ sr) \\
\Leftrightarrow \\
\quad (\exists\ ze: Z\_EXPR;\ oseqze: seq\ Z\_EXPR;\ seqr: seq\ \mathbb{R} \\
\quad \mid\ p \mapsto oseqze \in select\_from\_seq\_by\_param\ iseqze \\
\quad \bullet\ sr = SRVector\ ze \\
\qquad \land\ oze = ZSequence\ oseqze \\
\quad \lor\ sr = SRScalar\ ze \\
\qquad \land\ \langle oze \rangle = oseqze)
\end{array}
$$

The general case is then compounded as follows:

z

$$
\begin{array}{l}
\boldsymbol{selector\_expression}: (PARAMETER \times SPECIAL\_RESULT) \\
\quad \rightarrow (Z\_EXPR \nrightarrow Z\_EXPR) \\
\hline
\forall\ p: PARAMETER;\ sr: SPECIAL\_RESULT \bullet \\
\quad selector\_expression\ (p,\ sr) \\
\quad = display\_switch(\\
\qquad (ZSequence\ \fatsemi\ (selector\_expression\_from\_other\ sr)) \\
\qquad\quad \oplus\ selector\_expression\_from\_display\ (p,\ sr), \\
\qquad selector\_expression\_from\_other\ sr)
\end{array}
$$

This

z

$$
\begin{array}{|l}
\boldsymbol{virtualize\_selector}: LIB\_BLOCK\_VIRT\_FUN2 \\
\hline
\forall\ mvartypes: MVARTYPES;\ alb: A\_LIB\_BLOCK;\ mv\_ctxt: \mathbb{F}\ PVALUE \bullet \\
virtualize\_selector\ (mvartypes,\ alb,\ mv\_ctxt) = \\
(\mu lbvf1: LIB\_BLOCK\_VIRT\_FUN1 \\
\mid lbvf1 = \\
\qquad lbvf1\_inhibit \oplus \\
\qquad \{\ izemap: PVALUE \twoheadrightarrow\!\!\!\!\!\!+\ Z\_EXPR;\ used\_maskvars: \mathbb{F}\ PVALUE; \\
\qquad\ \ sr: SPECIAL\_RESULT;\ v: VIRTUAL;\ p: PARAMETER; \\
\qquad\ \ pt\_map: PVALUE \twoheadrightarrow\!\!\!\!\!\!+\ PNAME \times PORT\_TYPE;\ ize,\ oze: Z\_EXPR \\
\qquad \mid dom\ alb.port\_info.input\_port\_details = \{One\} \\
\qquad \land \{One \mapsto ize\} = izemap \\
\qquad \land pt\_map = \{One \mapsto (Elements, \\
\qquad\qquad\qquad (alb.port\_info.output\_port\_details\ \ One).port\_type)\} \\
\qquad \land (\{One \mapsto (p,\ sr)\},\ used\_maskvars) \\
\qquad\ \ = param\_sr\ (alb.block\_info.pars,\ mv\_ctxt,\ mvartypes)\ pt\_map \\
\qquad \land ize \mapsto oze \in selector\_expression\ (p,\ sr) \\
\qquad \land v = Virtual\ \{One \mapsto oze\} \\
\qquad \bullet\ izemap \mapsto (v,\ used\_maskvars) \\
\qquad \} \\
\bullet\ (lbvf1,\ \{One\}))
\end{array}
$$

6.7.10 Virtualize Terminator

z

$$
\begin{array}{|l}
\qquad \boldsymbol{virtualize\_terminator}: LIB\_BLOCK\_VIRT\_FUN2 \\
\hline
\qquad virtualize\_terminator = lbvf2\_triv
\end{array}
$$

6.7.11 Compounded Block Virtualizer

The following map assigns each block type supported for virtualization to its virtualization function.

z

$$\begin{array}{l}
\boldsymbol{virtualize\_table}: PVALUE \twoheadrightarrow\!\!\!\!\!\mapsto LIB\_BLOCK\_VIRT\_FUN2 \\
\hline
virtualize\_table = \{ \\
BusCreator \mapsto virtualize\_bus\_creator, \\
BusSelector \mapsto virtualize\_bus\_selector, \\
Constant \mapsto virtualize\_constant, \\
Demux \mapsto virtualize\_demux, \\
Mux \mapsto virtualize\_mux, \\
Selector \mapsto virtualize\_selector, \\
Terminator \mapsto virtualize\_terminator \\
\}
\end{array}$$

Using the table we now define a virtualizer for arbitrary library blocks. This is total since in the worst case it returns *virtualize_dummy1*.

z

$$\begin{array}{l}
\boldsymbol{virtualize\_libblock}: LIB\_BLOCK\_VIRT\_FUN2 \\
\hline
virtualize\_libblock = \\
\{mvartypes: MVARTYPES; alb: A\_LIB\_BLOCK; mv\_ctxt: \mathbb{P}\ PVALUE; \\
\ param: PARAM; bt: PVALUE \\
\mid param \in alb.block\_info.pars \\
\land param = (name \mathrel{\widehat=} BlockType, value \mathrel{\widehat=} bt) \\
\bullet (mvartypes, alb, mv\_ctxt) \mapsto \\
(((\lambda pv: PVALUE \bullet lbvf2\_inhibit) \oplus virtualize\_table)\ bt \\
\qquad (mvartypes, alb, mv\_ctxt))\}
\end{array}$$

6.7.12 Traversal Types

The following two function types generalise the above to deal with arbitrary blocks. Two additional results are required, the first an update to the block, the second a set of paths of blocks whose virtualization has been inhibited because of loop detection.

The parameter to the following is a map of input port names to Z expressions. The results are:

- the updated block (details of virtualization and of used mask variables updated as appropriate)
- the result of attempting the virtualization of this block as a *VIRTUAL*
- a set of used maskvariables
- a set of paths for blocks whose virtualisation has been inhibited because of loop detection

z

$$\boldsymbol{BLOCK\_VIRT\_FUN1} \mathrel{\widehat=}$$
$$(PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} Z\_EXPR)$$
$$\rightarrow A\_BLOCK \times VIRTUAL \times (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ PATH)$$

The parameters to the following are, first:

- the path of the block to be virtualized
- the types of the matlab variables in context
- the mask variables in context

then:

- the block to be virtualized

The result is a pair consisting of a function to complete the virtualization and the set of names of input ports whose translation is required to complete virtualization.

z

$$\boldsymbol{BLOCK\_VIRT\_FUN2} \mathrel{\widehat=}$$
$$PATH \times MVARTYPES \times (\mathbb{F}\ PVALUE) \rightarrow A\_BLOCK$$
$$\mathrel{+\!\!\!\!\rightarrow} BLOCK\_VIRT\_FUN1 \times \mathbb{F}\ PVALUE$$

The following is the type of a function which updates a subsystem by virtualising a block contained in the subsystem. This is in fact our "key function". Unlike the above functions, which expect to be supplied with expressions for the values on the input ports of the blocks, this kind of function is intended to find out this information for itself, if necessary virtualizing other blocks in the subsystem. It is therefore given the entire subsystem, which it updates and returns along with the specifics for the particular block whose virtualization was requested.

The parameters to the function are:

- the path of the subsystem containing the block to be virtualized
- the types of matlab variables
- a set of blocknames already visited (and hence whose output port values are to be considered unavailable, since their use would create a loop).

  If any block whose virtualisation is required is found to be dependent on one of these blocks (i.e. to have a significant input port connected to an output port of that block) then the virtualisation of that block is inhibited and its name reported in the result (for inclusion in a warning message).

then:

- the blockname of the block to be virtualized
- the blocknames of blocks which may not connect to this block without resulting in a virtual loop
- the subsystem containing the block to be virtualized

The function covers in principle the virtualisation of subsystem blocks, but in this version of ClawZ this always fails (with the useful side effect, however, of virtualizing as many as possible of the blocks within the subsystem)

The set of *PVALUES* returned by this function is the set of used mask variables. The set of paths returned is the set of paths of blocks whose virtualization was inhibited because of a loop in the wiring diagram, which is returned to permit a warning to be issued. Other reasons for failure to virtualize do not give rise to warnings or errors.

z

$$\boldsymbol{BLOCK\_VIRT\_FUN3} \mathrel{\widehat{=}} PATH \times MVARTYPES \times (\mathbb{F}\ PVALUE) \\ \rightarrow PVALUE \times (\mathbb{F}\ PVALUE) \times A\_SUBSYS \\ \nrightarrow VIRTUAL \times A\_SUBSYS \times (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ PATH)$$

This is the type of the key function by which all principal functions are parameterized.

6.7.13 Traversal Auxiliaries

The following are two functions for making the specific modifications to components of the model required in virtualization. [If these still only have single uses at the end then it might be better to inline them.]

z

$$\boldsymbol{vupdate\_bi}: (BLOCK\_INFO \times VIRTUAL \times \mathbb{F}\ PVALUE) \rightarrow BLOCK\_INFO$$

$$\forall\ bi: BLOCK\_INFO;\ virtual: VIRTUAL;\ used\_maskvars: \mathbb{P}\ PVALUE \bullet \\ vupdate\_bi\ (bi,\ virtual,\ used\_maskvars) \\ = (\quad pars \mathrel{\widehat{=}} bi.pars, \\ input\_port\_types \mathrel{\widehat{=}} bi.input\_port\_types, \\ output\_port\_types \mathrel{\widehat{=}} bi.output\_port\_types, \\ specification \mathrel{\widehat{=}} bi.specification, \\ invocation \mathrel{\widehat{=}} bi.invocation, \\ virtual \mathrel{\widehat{=}} virtual, \\ used\_maskvars \mathrel{\widehat{=}} used\_maskvars \\ )$$

z

$$
\begin{array}{l}
\textbf{\textit{vupdate\_alb}}: (A\_LIB\_BLOCK \times VIRTUAL \times \mathbb{F}\ PVALUE) \rightarrow A\_LIB\_BLOCK \\
\hline
\forall\ alb: A\_LIB\_BLOCK;\ virtual: VIRTUAL;\ used\_maskvars: \mathbb{P}\ PVALUE \bullet \\
vupdate\_alb\ (alb, virtual, used\_maskvars) \\
= (\quad block\_info \mathrel{\widehat{=}} vupdate\_bi\ (alb.block\_info, virtual, used\_maskvars), \\
\qquad port\_info \mathrel{\widehat{=}} alb.port\_info \\
)
\end{array}
$$

Note that in the following update specification the *used_maskvars* are supplemented rather than replaced by the relevant parameter.

z

$$
\begin{array}{l}
\textbf{\textit{vupdate\_si}}: (SUBSYS\_INFO \times VIRTUAL \times \mathbb{F}\ PVALUE) \rightarrow SUBSYS\_INFO \\
\hline
\forall\ si: SUBSYS\_INFO;\ virtual: VIRTUAL;\ used\_maskvars: \mathbb{F}\ PVALUE \bullet \\
vupdate\_si\ (si, virtual, used\_maskvars) \\
= (\quad subpars \mathrel{\widehat{=}} si.subpars, \\
\qquad syspars \mathrel{\widehat{=}} si.syspars, \\
\qquad lines \mathrel{\widehat{=}} si.lines, \\
\qquad specification \mathrel{\widehat{=}} si.specification, \\
\qquad invocation \mathrel{\widehat{=}} si.invocation, \\
\qquad virtual \mathrel{\widehat{=}} virtual, \\
\qquad used\_maskvars \mathrel{\widehat{=}} si.used\_maskvars \cup used\_maskvars, \\
\qquad mv\_ctxt \mathrel{\widehat{=}} si.mv\_ctxt, \\
\qquad action\_info \mathrel{\widehat{=}} si.action\_info \\
)
\end{array}
$$

z

$$
\begin{array}{l}
\textbf{\textit{vupdate\_ass}}: (A\_SUBSYS \times VIRTUAL \times \mathbb{F}\ PVALUE) \rightarrow A\_SUBSYS \\
\hline
\forall\ ass: A\_SUBSYS;\ virtual: VIRTUAL;\ used\_maskvars: \mathbb{P}\ PVALUE \bullet \\
vupdate\_ass\ (ass, virtual, used\_maskvars) \\
= (\quad subsys\_info \mathrel{\widehat{=}} vupdate\_si\ (ass.subsys\_info, virtual, used\_maskvars), \\
\qquad port\_info \mathrel{\widehat{=}} ass.port\_info, \\
\qquad blocks \mathrel{\widehat{=}} ass.blocks \\
)
\end{array}
$$

z

$$\begin{array}{l}
\textbf{\textit{vupdate\_ab}}: (A\_BLOCK \times VIRTUAL \times \mathbb{F}\ PVALUE) \to A\_BLOCK \\
\hline
\forall\ alb: A\_LIB\_BLOCK;\ virtual: VIRTUAL;\ used\_maskvars: \mathbb{P}\ PVALUE \bullet \\
vupdate\_ab\ (ALibBlock\ alb,\ virtual,\ used\_maskvars) \\
= ALibBlock\ (vupdate\_alb\ (alb,\ virtual,\ used\_maskvars)); \\
\\
\forall\ ass: A\_SUBSYS;\ virtual: VIRTUAL;\ used\_maskvars: \mathbb{P}\ PVALUE \bullet \\
vupdate\_ab\ (ASubsys\ ass,\ virtual,\ used\_maskvars) \\
= ASubsys\ (vupdate\_ass\ (ass,\ virtual,\ used\_maskvars))
\end{array}$$

The following is only intended for use where the blockname is the name of a block in the subsystem. As defined it is the identity function elsewhere.

z

$$\begin{array}{l}
\textbf{\textit{vupdate\_ab\_in\_ass}}: (A\_SUBSYS \times PVALUE \times VIRTUAL \times \mathbb{F}\ PVALUE) \\
\qquad \to A\_SUBSYS \\
\hline
\forall\ ass: A\_SUBSYS;\ ab: A\_BLOCK;\ bn: PVALUE; \\
\quad virtual: VIRTUAL;\ used\_maskvars: \mathbb{P}\ PVALUE \bullet \\
vupdate\_ab\_in\_ass\ (ass,\ bn,\ virtual,\ used\_maskvars) \\
= (subsys\_info \mathrel{\widehat{=}} ass.subsys\_info, \\
\quad port\_info \mathrel{\widehat{=}} ass.port\_info, \\
\quad blocks \mathrel{\widehat{=}} ass.blocks \oplus \\
\qquad \{m{:}\{bn \mapsto vupdate\_ab\ (ass.blocks\ bn,\ virtual,\ used\_maskvars)\} \\
\qquad \mid bn \in dom\ ass.blocks\})
\end{array}$$

z

$$\begin{array}{l}
\textbf{\textit{virtual}}: A\_BLOCK \to VIRTUAL \\
\hline
\forall\ as: A\_SUBSYS \bullet \\
virtual\ (ASubsys\ as) = as.subsys\_info.virtual; \\
\\
\forall\ alb: A\_LIB\_BLOCK \bullet \\
virtual\ (ALibBlock\ alb) = alb.block\_info.virtual
\end{array}$$

This is a function for making a *BLOCK_VIRT_FUN1* for a block which does not have virtual *VUnknown*.

z

```
│   $\textbf{nochange\_bvf1}: A\_BLOCK \rightarrow BLOCK\_VIRT\_FUN1$
├──────────────────────────
│   $\forall ab: A\_BLOCK \bullet$
│   $nochange\_bvf1\ ab = (\lambda\ emap: PVALUE \twoheadrightarrow Z\_EXPR \bullet$
│       $(ab, virtual\ ab, \{\}, \{\}))$
```

The following adapts *virtualize_libblock* to the form in which it is later required. It also imposes a check on the domain of the *emap* so that virtualization will fail unless all the requested input ports are available.

z

```
│   $\textbf{virtualize\_libblock2}: PATH \times MVARTYPES \times \mathbb{F}\ PVALUE$
│       $\rightarrow A\_LIB\_BLOCK \rightarrow BLOCK\_VIRT\_FUN1 \times \mathbb{F}\ PVALUE$
├──────────────────────────
│ $\forall path: PATH; mvartypes: MVARTYPES; mv\_ctxt: \mathbb{P}\ PVALUE;$
│     $alb: A\_LIB\_BLOCK \bullet$
│ $virtualize\_libblock2\ (path, mvartypes, mv\_ctxt)\ alb =$
│     $if\ alb.block\_info.virtual = VUnknown$
│     $then\ (\mu\ lbvf1: LIB\_BLOCK\_VIRT\_FUN1; inports: \mathbb{F}\ PVALUE;$
│               $bvf1: BLOCK\_VIRT\_FUN1$
│             $|\ (lbvf1, inports) = virtualize\_libblock\ (mvartypes, alb, mv\_ctxt)$
│             $\land\ bvf1 =$
│               $\{emap: PVALUE \twoheadrightarrow Z\_EXPR; virtual': VIRTUAL;$
│                     $used\_mvs: \mathbb{F}\ PVALUE$
│               $|\ dom\ emap = inports \land (virtual', used\_mvs) = lbvf1\ emap$
│               $\lor\ dom\ emap \neq inports \land (virtual', used\_mvs) = (VInhibit, \{\})$
│               $\bullet\ emap \mapsto (ALibBlock\ ($
│                         $block\_info \mathrel{\widehat=} vupdate\_bi\ (alb.block\_info, virtual', used\_mvs),$
│                         $port\_info \mathrel{\widehat=} alb.port\_info),$
│                     $virtual', used\_mvs, \{\})\}$
│             $\bullet\ (bvf1, inports))$
│     $else\ (nochange\_bvf1\ (ALibBlock\ alb), \{\})$
```

The following function takes:

- the set of lines of some subsystem
- the name of a block in the subsystem and a set of input portnames on that block

and delivers a sequence of blocknames and map from input portname to block, output portname pairs.

It is to be used to determine which blocks must be virtualized before some other block to be virtualized.

z

$$\begin{array}{l}
\boldsymbol{blocks\_from\_ports}: \mathbb{F}\ LINE \rightarrow (PVALUE \times \mathbb{F}\ PVALUE) \\
\quad \rightarrow ((seq\ PVALUE) \times (PVALUE \twoheadrightarrow PVALUE \times PVALUE)) \\
\hline
\forall\ lines: \mathbb{F}\ LINE;\ bn: PVALUE;\ portnames: \mathbb{F}\ PVALUE; \\
\quad portmap: PVALUE \twoheadrightarrow PVALUE \times PVALUE \\
\mid portmap = \{ip: portnames;\ line: lines \\
\qquad \mid (block \mathrel{\widehat{=}} bn,\ port \mathrel{\widehat{=}} ip) \in line.destinations \\
\qquad \bullet\ ip \mapsto (line.source.block,\ line.source.port)\} \\
\bullet\ blocks\_from\_ports\ lines\ (bn,\ portnames) = \\
\quad ((\mu\ spv: iseq\ PVALUE \mid ran\ spv = dom\ (ran\ portmap)),\ portmap)
\end{array}$$

The following function takes:

- a map of *VIRTUAL*s obtained by virtualizing a set of blocks
- a map from input portname to block, output portname pairs, indicating which input ports (for some other target block) are connected to output ports of blocks in the domain of the first parameter

and returns a map of expressions for the input ports of the target block.

If a port is connected to a virtualized block and the *VIRTUAL* for that block includes an entry for the relevant output port (which *should* always be the case), then that entry is used for the port. If a port is connected to a block whose *VIRTUAL* status is *VInhibit* or *VUnknown* (which latter should never happen) then an expression consisting of the relevant port name selected from the local blockname is used.

z

$$\begin{array}{l}
\boldsymbol{emap\_from\_virtual\_map}: (PATH \times A\_SUBSYS \times (PVALUE \twoheadrightarrow VIRTUAL)) \\
\quad \rightarrow (PVALUE \twoheadrightarrow PVALUE \times PVALUE) \rightarrow (PVALUE \twoheadrightarrow Z\_EXPR) \\
\hline
\forall\ path: PATH;\ ass: A\_SUBSYS;\ vmap: PVALUE \twoheadrightarrow VIRTUAL; \\
\quad ipmap: PVALUE \twoheadrightarrow PVALUE \times PVALUE \bullet \\
emap\_from\_virtual\_map\ (path,\ ass,\ vmap)\ ipmap = \\
\qquad \{ip: PVALUE;\ ze: Z\_EXPR;\ opmap: PVALUE \twoheadrightarrow Z\_EXPR; \\
\qquad\ bn,\ pn: PVALUE \\
\qquad \mid ip \mapsto (bn,\ pn) \in ipmap \\
\qquad \wedge\ (bn \mapsto Virtual\ opmap \in vmap \\
\qquad\quad \wedge\ pn \mapsto ze \in opmap \\
\qquad\quad \vee\ \neg(\exists opmap2: PVALUE \twoheadrightarrow Z\_EXPR \bullet\ bn \mapsto Virtual\ opmap2 \in vmap) \\
\qquad\quad \wedge\ ze = outport\_ident\ path\ ass.blocks\ (block \mathrel{\widehat{=}} bn,\ port \mathrel{\widehat{=}} pn)) \\
\qquad \bullet\ ip \mapsto ze\}
\end{array}$$

6.7.14 Principal Functions

The following function recursively maps a *BLOCK_VIRT_FUN3* over a set of blocks in some subsystem (and their subsystems).

z

$$\begin{array}{l}
\boldsymbol{subsys\_map\_bvf3}: BLOCK\_VIRT\_FUN3 \\
\quad \rightarrow (PATH \times MVARTYPES \times \mathbb{F}\ PVALUE) \\
\quad \rightarrow (A\_SUBSYS \times (\mathbb{F}\ PVALUE) \times seq\ PVALUE) \\
\quad \rightarrow (A\_SUBSYS \times (PVALUE \twoheadrightarrow VIRTUAL) \times (\mathbb{F}\ PVALUE) \times \mathbb{F}\ PATH) \\
\hline
\forall\ bvf3: BLOCK\_VIRT\_FUN3; \\
\quad path: PATH;\ mvartypes: MVARTYPES;\ mv\_ctxt: \mathbb{F}\ PVALUE; \\
\quad ass: A\_SUBSYS;\ lblocks: \mathbb{F}\ PVALUE;\ blocks: seq\ PVALUE;\ block: PVALUE \\
\bullet\ subsys\_map\_bvf3\ bvf3\ (path,\ mvartypes,\ mv\_ctxt)\ (ass,\ lblocks,\ \langle\rangle) \\
\qquad = (ass,\ \{\},\ \{\},\ \{\}) \\
\land\ subsys\_map\_bvf3\ bvf3\ (path,\ mvartypes,\ mv\_ctxt)\ (ass,\ lblocks,\ \langle block\rangle \frown blocks) \\
\qquad = (\mu\ ass',\ ass'': A\_SUBSYS;\ vmap,\ vmap': PVALUE \twoheadrightarrow VIRTUAL; \\
\qquad\qquad used\_mvs,\ used\_mvs': \mathbb{F}\ PVALUE;\ fpath,\ fpath': \mathbb{F}\ PATH;\ virtual: VIRTUAL \\
\qquad\quad \mid (virtual,\ ass',\ used\_mvs,\ fpath) \\
\qquad\qquad = bvf3\ (path,\ mvartypes,\ mv\_ctxt)\ (block,\ lblocks, ass) \\
\qquad\quad \land\ (ass'',\ vmap,\ used\_mvs',\ fpath') \\
\qquad\qquad = subsys\_map\_bvf3\ bvf3\ (path,\ mvartypes,\ mv\_ctxt)\ (ass',\ lblocks,\ blocks) \\
\qquad\quad \bullet\ (ass'',\ vmap \cup \{block \mapsto virtual\},\ used\_mvs \cup used\_mvs',\ fpath \cup fpath'))
\end{array}$$

The following specifies how to virtualize a subsystem given a *BLOCK_VIRT_FUN3*. The specification is written on the assumption that the order of processing blocks does not matter. This may be the case, but it should be noted that the order may affect the output from clawz if blocks have their virtualization inhibited by loop detection. For example the set of blocks inhibited may depend on the order in which blocks are processed. To give consistency the blocks should be processed in the same order as they appear in the list representation of the block map (which the implementation sorts earlier).

As curently specified no attempt is made to virtualize the subsystem itself. Only library blocks within the subsystem are liable to be virtualized (the *bvf3* is mapped over all blocks in the system, but since it will use this function to do the subsystems this will only virtualize library blocks in the subsystems).

z

```
    virtualize_subsys: BLOCK_VIRT_FUN3
        → PATH × MVARTYPES × ℙ PVALUE
        → A_SUBSYS → BLOCK_VIRT_FUN1 × 𝔽 PVALUE
────────────────────────
∀ bvf3: BLOCK_VIRT_FUN3;
  path: PATH; mvartypes: MVARTYPES; mv_ctxt: ℙ PVALUE;
  ass: A_SUBSYS•
virtualize_subsys bvf3 (path, mvartypes, mv_ctxt) ass
= if ass.subsys_info.virtual = VUnknown
  then (μ bvf1: BLOCK_VIRT_FUN1; ass′: A_SUBSYS; vmap: PVALUE ⇸ VIRTUAL;
          used_maskvars: 𝔽 PVALUE; fpaths: 𝔽 PATH
      | (ass′, vmap, used_maskvars, fpaths)
        = subsys_map_bvf3 bvf3 (path, mvartypes, mv_ctxt)
            (ass, {}, (μ bnames: seq PVALUE | ran bnames = dom ass.blocks))
      ∧ bvf1 =
            {emap: PVALUE ⇸ Z_EXPR
            • emap ↦ (ASubsys (
                subsys_info ≙ vupdate_si (ass′.subsys_info, VInhibit, used_maskvars),
                blocks ≙ ass′.blocks,
                port_info ≙ ass′.port_info),
              VInhibit, used_maskvars, fpaths)}
      • (bvf1, {}))
  else (nochange_bvf1 (ASubsys ass), {})
```

Subsystem and library block virtualization are now combined to specify a virtualizer for all blocks.

z

```
    virtualize_block_p: BLOCK_VIRT_FUN3 → BLOCK_VIRT_FUN2
────────────────────────
∀ bvf3: BLOCK_VIRT_FUN3;
  path: PATH; mvartypes: MVARTYPES; mv_ctxt: 𝔽 PVALUE;
  ab: A_BLOCK
•
      (∃ ass: A_SUBSYS | ASubsys ass = ab •
      virtualize_block_p bvf3 (path, mvartypes, mv_ctxt) ab
      = virtualize_subsys bvf3 (path, mvartypes, mv_ctxt) ass)
∧
      (∃ alb: A_LIB_BLOCK | ALibBlock alb = ab •
      virtualize_block_p bvf3 (path, mvartypes, mv_ctxt) ab
      = virtualize_libblock2 (path, mvartypes, mv_ctxt) alb)
```

6.7.15 The Key Function

We now specify the required *BLOCK_VIRT_FUN3* key function. This virtualizes a block by recursively virtualizing the blocks connected to its input ports and then transforming the input expressions in a manner specific to the block.

In the following specification of the key function *virtualize_block_in_subsys*:

blocknames' is the set of all blocknames involved in the recursion in the current subsystem, and is used for virtual loop detection.

inportnames is the list of input port names whose values are required to virtualize the block.

blockseq is the list of blocks which must supply the expressions required for the input ports.

portmap shows the relationship between the required input ports and the block/ouputport where it is to be found.

vmap is the map of virtuals obtained by virtualizing all the blocks in blockseq.

emap is the Z expression map derived from vmap and portmap.

fpaths is a set of paths of blocks whose virtualization has been inhibited in this subsystem

fpaths2 is a set of paths of blocks whose virtualization has been inhibited in the block being virtualized (always empty if it is a library block).

z

$$
\begin{array}{l}
\boldsymbol{virtualize\_block\_in\_subsys}: BLOCK\_VIRT\_FUN3 \\
\hline
\forall\ path: PATH;\ mvartypes: MVARTYPES; \\
\quad blocknames,\ blocknames': \mathbb{F}\ PVALUE; \\
\quad bn: PVALUE;\ ass: A\_SUBSYS;\ bvf1: BLOCK\_VIRT\_FUN1; \\
\quad inportnames: \mathbb{F}\ PVALUE;\ lines: \mathbb{F}\ LINE;\ blockseq: seq\ PVALUE; \\
\quad portmap: PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} PVALUE \times PVALUE; \\
\quad maskvars: \mathbb{F}\ PVALUE \\
\mid maskvars = ass.subsys\_info.mv\_ctxt \\
\land blocknames' = blocknames \cup \{bn\} \\
\land (bvf1,\ inportnames) = virtualize\_block\_p\ virtualize\_block\_in\_subsys \\
\qquad\quad (path \frown \langle bn \rangle,\ mvartypes,\ maskvars)\ (ass.blocks\ \ bn) \\
\land lines = ass.subsys\_info.lines \\
\land (blockseq,\ portmap) = blocks\_from\_ports\ lines\ (bn,\ inportnames) \\
\bullet\ virtualize\_block\_in\_subsys\ (path,\ mvartypes,\ blocknames)\ (bn,\ blocknames,\ ass) \\
=\ if\ \ blocknames' \cap ran\ \ blockseq = \{\} \\
\quad then \\
\quad (\mu\ ass',\ ass'': A\_SUBSYS;\ vmap: PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} VIRTUAL; \\
\qquad used\_mvs1,\ used\_mvs2: \mathbb{F}\ PVALUE;\ ab: A\_BLOCK; \\
\qquad virtual: VIRTUAL;\ fpaths,\ fpaths2: \mathbb{F}\ PATH; \\
\qquad blocks': PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} A\_BLOCK;\ emap: PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} Z\_EXPR \\
\quad \mid (ass',\ vmap,\ used\_mvs1,\ fpaths) = subsys\_map\_bvf3\ virtualize\_block\_in\_subsys \\
\qquad\quad (path,\ mvartypes,\ maskvars)\ (ass,\ blocknames',\ blockseq) \\
\quad \land emap = emap\_from\_virtual\_map\ (path,\ ass',\ vmap)\ portmap \\
\quad \land (ab,\ virtual,\ used\_mvs2,\ fpaths2) = bvf1\ \ emap \\
\quad \land blocks' = ass'.blocks \oplus \{bn \mapsto ab\} \\
\quad \land ass'' = (subsys\_info \mathrel{\widehat{=}} ass'.subsys\_info, \\
\qquad\qquad\qquad port\_info \mathrel{\widehat{=}} ass'.port\_info,\ blocks \mathrel{\widehat{=}} blocks') \\
\quad \bullet\ (virtual,\ ass'',\ used\_mvs1 \cup used\_mvs2\ ,\ fpaths \cup fpaths2) \\
\quad) \\
\quad else \\
\quad (\mu\ ass': A\_SUBSYS \\
\quad \mid ass' = vupdate\_ab\_in\_ass\ (ass,\ bn,\ VInhibit,\ \{\}) \\
\quad \bullet\ (VInhibit,\ ass',\ \{\},\ \{path \frown \langle bn \rangle\}) \\
\quad)
\end{array}
$$

We can now define non-parameterised versions of the required principal functions:

z

$\boldsymbol{virtualize\_block}: BLOCK\_VIRT\_FUN2$

$virtualize\_block = virtualize\_block\_p\ virtualize\_block\_in\_subsys$

6.7.16 Virtualize System

Note that in the following function the set of paths *fpaths* should be reported by the implementation in a warning. These are the paths of blocks whose virtualization was inhibited to break virtual cycles in the wiring diagram.

z

$\boldsymbol{a\_block\_name}: A\_BLOCK \nrightarrow PVALUE$

$\forall\ ab: A\_BLOCK;\ pv: PVALUE \bullet$
$\quad ab \mapsto pv \in a\_block\_name$
$\Leftrightarrow$
$\quad (\exists\ alb: A\_LIB\_BLOCK \bullet$
$\quad ab = ALibBlock\ alb$
$\quad \land (name \mathrel{\widehat{=}} Name,\ value \mathrel{\widehat{=}} pv) \in alb.block\_info.pars)$

$\lor \quad (\exists\ ass: A\_SUBSYS \bullet$
$\quad ab = ASubsys\ ass$
$\quad \land (name \mathrel{\widehat{=}} Name,\ value \mathrel{\widehat{=}} pv) \in ass.subsys\_info.syspars)$

z

$\boldsymbol{virtualize\_system}: MVARTYPES \to A\_BLOCK \to A\_BLOCK$

$\forall\ mvartypes: MVARTYPES;\ ab,\ ab': A\_BLOCK;$
$\quad sys\_name: PVALUE;\ bvf1: BLOCK\_VIRT\_FUN1;$
$\quad rb,\ used\_mvars: \mathbb{F}\ PVALUE;\ v: VIRTUAL;\ fpaths: \mathbb{F}\ PATH$
$\mid ab \mapsto sys\_name \in a\_block\_name$
$\land (bvf1,\ rb) = virtualize\_block\ (\langle sys\_name \rangle,\ mvartypes,\ \{\})\ ab$
$\land (ab',\ v,\ used\_mvars,\ fpaths) = bvf1\ \{\}$
$\bullet\ virtualize\_system\ mvartypes\ ab = ab'$

6.8 Block Synthesis

6.8.1 Invocation Sorting

The following specifies the order in which declarations of a schema are to be output. It applies both to synthesised blocks and to subsystem schemas.

The following ordering is imposed on the declarations:

1. action then enable then trigger ports, if present
2. other input ports in ascending numeric order
3. other blocks in ascii ordering of local z name
4. output ports in ascending numeric order

This ordering is specified as a relation over INVKEY and then lifted over INVOCATIONs.

identbefore is to be understood as the reflexive ascii lexicographic ordering on Z identifiers.

z

$$\boldsymbol{identbefore}: IDENT \leftrightarrow IDENT$$

$$true$$

z

$$\boldsymbol{keybefore}: INVKEY \leftrightarrow INVKEY$$

$$\begin{array}{l}
keybefore = \\
\{lik,\ rik: INVKEY;\ n1,\ n2: \mathbb{N};\ l1,\ l2: IDENT \\
\mid n1 \leq n2 \land (l1,\ l2) \in identbefore \land \\
(lik = ActionInv\ n1 \land rik = ActionInv\ n2 \\
\lor lik = ActionInv\ n1 \land rik = EnableInv \\
\lor lik = EnableInv \land rik = TriggerInv \\
\lor lik = TriggerInv \land rik = InportInv\ n1 \\
\lor lik = InportInv\ n1 \land rik = InportInv\ n2 \\
\lor lik = InportInv\ n1 \land rik = OtherInv\ l1 \\
\lor lik = OtherInv\ l1 \land rik = OtherInv\ l2 \\
\lor lik = OtherInv\ l1 \land rik = OutportInv\ n1 \\
\lor lik = OutportInv\ n1 \land rik = OutportInv\ n2) \\
\bullet (lik,\ rik)\}^*
\end{array}$$

We now specify the required ordering on invocations by a function which maps sets of *INVKEY* tagged values to ordered sequences (an invocation is one example of such a tagged value).

z

$[X]$

$$\textbf{\textit{sort\_by\_invkey}}: \mathbb{F}\ (INVKEY \times X) \rightarrow seq\ (INVKEY \times X)$$

$$\begin{array}{l}
\forall\ invset: \mathbb{F}\ (INVKEY \times X);\ invseq: seq\ (INVKEY \times X) \bullet \\
sort\_by\_invkey\ invset = invseq \\
\Rightarrow invset = ran\ invseq \\
\quad \wedge \quad (\forall\ n1, n2: dom\ invseq;\ fi1,\ fi2: INVKEY \\
\qquad\quad \mid n1 < n2 \wedge fi1 = first\ (invseq\ n1) \wedge\ fi2 = first\ (invseq\ n2) \\
\qquad\quad \bullet\ fi1 \neq fi2 \wedge (fi1,\ fi2) \in keybefore)
\end{array}$$

6.8.2 Synthesis Preliminaries

Block synthesis traverses the model looking for instances of synthesizable blocks. It then fills in the *specification* and *invocation* fields for these blocks.

Following the established pattern we now address functions which synthesize specifications for certain kinds of block. These will usually be specified the following type.

z

$$\begin{array}{l}
\textbf{\textit{BLOCK\_SYN\_FUN2}} \mathrel{\widehat{=}} PATH \times \mathbb{P}\ PVALUE \times A\_LIB\_BLOCK \times A\_SUBSYS \\
\qquad\qquad \mathrel{+\!\!\!\rightarrow} ((Z\_SPEC \times (Z\_DECL \times Z\_DECL \times Z\_DECL)) \times (\mathbb{F}\ PVALUE))
\end{array}$$

In the argument above the path is the pathname of the block to be synthesized and the set of *PVALUE*s is the set of masked variables in the context of that block, i.e. the set of names which should be treated as local variables rather than as the names of values set in a matlab .m file. The *A_SUBSYS* is the smallest enclosing subsystem. The returned set of *PVALUE*s is the set of masked variables which were used in parameters translated into Z.

The synthesis functions will be required with the following type:

z

$$\begin{array}{l}
\textbf{\textit{BLOCK\_SYN\_FUN}} \mathrel{\widehat{=}} PATH \times \mathbb{P}\ PVALUE \times A\_SUBSYS \\
\qquad\qquad \rightarrow A\_BLOCK \mathrel{+\!\!\!\rightarrow} A\_BLOCK
\end{array}$$

In the above the path is the pathname of the block to be synthesised and the set of *PVALUE*s is the set of masked variables in the context of that block, and the subsystem is the smallest enclosing subsystem.

Conversion from one to the other is accomplished as follows:

z

$$
\begin{array}{|l}
\quad \boldsymbol{lift\_bsf}: BLOCK\_SYN\_FUN2 \to BLOCK\_SYN\_FUN \\
\hline
\forall\ bsf2: BLOCK\_SYN\_FUN2;\ path: PATH;\ mv\_ctxt: \mathbb{F}\ PVALUE;\ ass: A\_SUBSYS \bullet \\
\quad lift\_bsf\ bsf2\ (path, mv\_ctxt, ass) = \\
\quad \{ \quad ab, ab': A\_BLOCK;\ alb, alb': A\_LIB\_BLOCK;\ pars: \mathbb{F}\ PARAM; \\
\quad\quad zspec: Z\_SPEC;\ zdecls: Z\_DECL \times Z\_DECL \times Z\_DECL; \\
\quad\quad used\_mvs: \mathbb{F}\ PVALUE \\
\quad | \quad ALibBlock\ alb = ab \land ALibBlock\ alb' = ab' \\
\quad \land \quad ((zspec, zdecls), used\_mvs) = bsf2\ (path, mv\_ctxt, alb, ass) \\
\quad \land \quad alb' = (\\
\quad\quad\quad block\_info \mathrel{\widehat=} (pars \mathrel{\widehat=} alb.block\_info.pars, \\
\quad\quad\quad\quad input\_port\_types \mathrel{\widehat=} alb.block\_info.input\_port\_types, \\
\quad\quad\quad\quad output\_port\_types \mathrel{\widehat=} alb.block\_info.output\_port\_types, \\
\quad\quad\quad\quad specification \mathrel{\widehat=} zspec, \\
\quad\quad\quad\quad invocation \mathrel{\widehat=} (OtherInv\ (path2loci\ path), zdecls), \\
\quad\quad\quad\quad virtual \mathrel{\widehat=} VInhibit, \\
\quad\quad\quad\quad used\_maskvars \mathrel{\widehat=} used\_mvs), \\
\quad\quad\quad port\_info \mathrel{\widehat=} alb.port\_info) \\
\quad \bullet \quad ab \mapsto ab'\}
\end{array}
$$

The core block-type-specific part of synthesis is computation of the predicate for the schema. Construction of the full specification and the method of invocation of this specification follows a general pattern specified below as *wrap_predicate*.

Preliminary to that we show how the declarations for the schema are determined by the *PORT_INFO* for the synthesized block.

z

$$
\begin{array}{|l}
\quad \boldsymbol{port\_invocations}: PORT\_INFO \to \mathbb{F}\ INVOCATION \\
\hline
\quad \forall\ pi: PORT\_INFO \bullet \\
\quad port\_invocations\ pi = \\
\quad\ \ (inport\_block\_invocation\ (\!|dom\ pi.input\_port\_details|\!)) \\
\quad \cup\ (outport\_block\_invocation\ (\!|dom\ pi.output\_port\_details|\!))
\end{array}
$$

Note that the *main_inv* is used here but that this is still OK for making the declaration part of synthesized state held and reset schemas because these always have the same declarations as the main schema. It would not do for state held and reset schemas for subsystems.

z

$$\boldsymbol{zdecl\_from\_invs}: \mathbb{F}\ INVOCATION \rightarrow Z\_DECL$$

$$\forall\ invs: \mathbb{F}\ INVOCATION \bullet$$
$$zdecl\_from\_invs\ invs = {}^\frown/\ (sort\_by\_invkey\ invs \fatsemi second \fatsemi main\_inv)$$

z

$$\boldsymbol{port\_declarations}: PORT\_INFO \rightarrow Z\_DECL$$

$$\forall\ pi: PORT\_INFO \bullet$$
$$port\_declarations\ pi = zdecl\_from\_invs\ (port\_invocations\ pi)$$

The following specification gives the invocation for an element of the state in a synthesized block.

z

$$\boldsymbol{state\_invocation}: PVALUE \rightarrow INVOCATION$$

$$\begin{aligned}
&\forall pv: PVALUE \bullet\\
&state\_invocation\ pv = (OtherInv\ (pvalue2ident\ pv),\\
&\qquad\qquad (\langle DecDec\ (\\
&\qquad\qquad\qquad names \mathrel{\widehat=} \langle pvalue2ident\ pv\rangle,\\
&\qquad\qquad\qquad type \mathrel{\widehat=} Ident\ Ui)\rangle,\ \langle\rangle,\ \langle\rangle))
\end{aligned}$$

The following specifies the method of constructing the specification and its invocation when no maskvariables have been used in the predicate.

z

$$\begin{aligned}
&\boldsymbol{wrap\_predicate\_nomv}: (WORD \times IDENT)\\
&\rightarrow (Z\_DECL \times Z\_PRED) \rightarrow (Z\_SPEC \times Z\_DECL)
\end{aligned}$$

$$\begin{aligned}
&\forall\ i: IDENT;\ pr: Z\_PRED;\\
&\qquad defname: WORD;\ spec: Z\_SPEC;\ decl,\ inv: Z\_DECL\\
&\mid spec = \langle SchemaBox\ (name \mathrel{\widehat=} defname,\ decl \mathrel{\widehat=} decl,\ pred \mathrel{\widehat=} pr)\rangle\\
&\land\ inv = \langle DecDec\ (names \mathrel{\widehat=} \langle i\rangle,\ type \mathrel{\widehat=} SchemaRef\ defname)\rangle\\
&\bullet\ wrap\_predicate\_nomv\ (defname,\ i)\ (decl,\ pr) = (spec,\ inv)
\end{aligned}$$

The more complex case involving maskvariables is as follows. This version should only be used with a non-empty set of used maskvariables.

z

$$\begin{array}{l}
\boldsymbol{wrap\_predicate\_mv}: (WORD \times IDENT \times \mathbb{F}\ PVALUE)\\
\to (Z\_DECL \times Z\_PRED) \to (Z\_SPEC \times Z\_DECL)\\
\hline
\forall i: IDENT;\ used\_maskvars: \mathbb{F}\ PVALUE;\\
\quad pr: Z\_PRED;\ defname: WORD;\ spec: Z\_SPEC;\ decl,\ inv: Z\_DECL\\
\mid spec = \langle AbbrevDef(\\
\qquad\qquad ident \mathrel{\widehat=} word2ident\ defname,\\
\qquad\qquad value \mathrel{\widehat=} make\_hschema\_abstraction\ used\_maskvars\ (\\
\qquad\qquad\qquad\qquad ZHSchema(decl \mathrel{\widehat=} decl,\ pred \mathrel{\widehat=} pr)))\rangle\\
\land inv = \langle make\_invocation\ defname\ i\ used\_maskvars\rangle\\
\bullet\ wrap\_predicate\_mv\ (defname,\ i,\ used\_maskvars)\ (decl,\ pr) = (spec,\ inv)
\end{array}$$

This version can be used in ignorance of whether the set of used maskvariables is empty. The "_wi" suffix distinguishes this version which requires a word and an identifier from the previous interface which required a path. This version is used for state hold and reset schema synthesis where suffixed need to be added to the names.

z

$$\begin{array}{l}
\boldsymbol{wrap\_predicate\_wi}: (WORD \times IDENT \times \mathbb{F}\ PVALUE)\\
\to (Z\_DECL \times Z\_PRED) \to (Z\_SPEC \times Z\_DECL)\\
\hline
\forall w: WORD;\ i: IDENT;\ used\_maskvars: \mathbb{F}\ PVALUE;\ zdecl: Z\_DECL;\ pr: Z\_PRED \bullet\\
\qquad wrap\_predicate\_wi\ (w,\ i,\ used\_maskvars)\ (zdecl,\ pr) =\\
\qquad if\ used\_maskvars = \{\}\\
\qquad then\ wrap\_predicate\_nomv\ (w,\ i)\ (zdecl,\ pr)\\
\qquad else\ wrap\_predicate\_mv\ (w,\ i,\ used\_maskvars)\ (zdecl,\ pr)
\end{array}$$

This version supports the original interface in which a path is supplied instead of the global and local names.

z

$$\begin{array}{l}
\boldsymbol{wrap\_predicate}: (PATH \times \mathbb{F}\ PVALUE) \to (Z\_DECL \times Z\_PRED)\\
\qquad\qquad\qquad \to (Z\_SPEC \times Z\_DECL)\\
\hline
\forall path: PATH;\ used\_maskvars: \mathbb{F}\ PVALUE;\ zdecl: Z\_DECL;\ pr: Z\_PRED \bullet\\
\qquad wrap\_predicate\ (path,\ used\_maskvars)\ (zdecl,\ pr) =\\
\qquad wrap\_predicate\_wi\ (path2globw\ path,\ path2loci\ path,\ used\_maskvars)\ (zdecl,\ pr)
\end{array}$$

We now specify how hold and reset schemas are prepared. These schemas must have exactly the same declaration part as the plain schema which they are associated with, so this declaration part is required as a parameter. In order to construct the predicate information about the names of the state components is required. This is to be presented as a set of triples of names. Each triple consists of the name of a state component, the name of the corresponding after state and the name of the corresponding initial state component. These could be generated from the one name, but maybe we shouldn't wire in the conventions which allow this to be done.

z

$$
\begin{array}{l}
\boldsymbol{state\_hold\_schema}: \\
(WORD \times IDENT \times (\mathbb{F}\ PVALUE) \times Z\_DECL \\
\quad \times \mathbb{F}\ (PVALUE \times PVALUE \times PVALUE)) \\
\to (Z\_SPEC \times Z\_DECL) \\
\hline
\forall w: WORD;\ i: IDENT;\ used\_maskvars: \mathbb{F}\ PVALUE;\ zdecl: Z\_DECL; \\
\ names: \mathbb{F}\ (PVALUE \times PVALUE \times PVALUE) \bullet \\
\quad state\_hold\_schema\ (w,\ i,\ used\_maskvars,\ zdecl,\ names) = \\
\quad wrap\_predicate\_wi\ (w_r\ w,\ i,\ used\_maskvars) \\
\qquad (zdecl, \\
\qquad PredConj \\
\qquad\quad \{\ tr: names;\ s,\ sp,\ is: PVALUE \\
\qquad\quad \mid tr = (s,\ sp,\ is) \\
\qquad\quad \bullet\ PredEq\ (Ident\ (pvalue2ident\ s),\ \{Ident\ (pvalue2ident\ sp)\})\} \\
\qquad)
\end{array}
$$

z

$$
\begin{array}{l}
\boldsymbol{state\_reset\_schema}: \\
(WORD \times IDENT \times (\mathbb{F}\ PVALUE) \times Z\_DECL \\
\quad \times \mathbb{F}\ (PVALUE \times PVALUE \times PVALUE)) \\
\to (Z\_SPEC \times Z\_DECL) \\
\hline
\forall w: WORD;\ i: IDENT;\ used\_maskvars: \mathbb{F}\ PVALUE;\ zdecl: Z\_DECL; \\
\ names: \mathbb{F}\ (PVALUE \times PVALUE \times PVALUE) \bullet \\
\quad state\_reset\_schema\ (w,\ i,\ used\_maskvars,\ zdecl,\ names) = \\
\quad wrap\_predicate\_wi\ (w_r\ w,\ i,\ used\_maskvars) \\
\qquad (zdecl, \\
\qquad PredConj \\
\qquad\quad \{\ tr: names;\ s,\ sp,\ is: PVALUE \\
\qquad\quad \mid tr = (s,\ sp,\ is) \\
\qquad\quad \bullet\ PredEq\ (Ident\ (pvalue2ident\ is),\ \{Ident\ (pvalue2ident\ sp)\})\ \} \\
\qquad)
\end{array}
$$

This version of *wrap_predicate* is to be used for synthesis of blocks which have state. It returns up to three schemas according to the setting of the hold context.

The parameter is a tuple with the following components:

- The full path of the block being synthesized.

- The mask variables used in the synthesized block predicate.
- The declaration part of the block schema.
- The predicate part of the block schema.
- A set of triples of state component names: (before, after, initial).

z

$$
\begin{array}{l}
\boldsymbol{state\_wrap\_predicate}: \\
(PATH \times (\mathbb{F}\ PVALUE) \times Z\_DECL \times Z\_PRED \\
\quad \times HOLD\_CONTEXT \times \mathbb{F}\ (PVALUE \times PVALUE \times PVALUE)) \\
\to (Z\_SPEC \times (Z\_DECL \times Z\_DECL \times Z\_DECL)) \\
\hline
\forall path: PATH;\ used\_maskvars: \mathbb{F}\ PVALUE;\ zdecl: Z\_DECL;\ zpred: Z\_PRED; \\
\ names: \mathbb{F}\ (PVALUE \times PVALUE \times PVALUE);\ hc: HOLD\_CONTEXT \bullet \\
\quad state\_wrap\_predicate\ (path,\ used\_maskvars,\ zdecl,\ zpred,\ hc,\ names) = \\
\quad (\mu\ zs,\ zsh,\ zsr: Z\_SPEC;\ zd,\ zdh,\ zdr: Z\_DECL;\ w: WORD;\ i: IDENT \\
\quad \mid w = path2globw\ path \\
\quad \land\ i = path2loci\ path \\
\quad \land\ (zs,\ zd) = wrap\_predicate\ (path,\ used\_maskvars)\ (zdecl,\ zpred) \\
\quad \land\ (zsh,\ zdh) = \\
\qquad if\ hc \in \{HCHeld,\ HCUnknown\} \\
\qquad then\ state\_hold\_schema\ (w,\ i,\ used\_maskvars,\ zdecl,\ names) \\
\qquad else\ (\langle\rangle,\ \langle\rangle) \\
\quad \land\ (zsr,\ zdr) = \\
\qquad if\ hc \in \{HCReset,\ HCUnknown\} \\
\qquad then\ state\_reset\_schema\ (w,\ i,\ used\_maskvars,\ zdecl,\ names) \\
\qquad else\ (\langle\rangle,\ \langle\rangle) \\
\quad \bullet\ (zs \frown zsh \frown zsr,\ (zd,\ zdh,\ zdr)) \\
\quad)
\end{array}
$$

6.8.3 Synthesize Bus Creator and Mux

Note that this specification shows no check on the block type, and should therefore be used only when the block type is known.

z

$$\begin{array}{l}
\boldsymbol{synthesize\_bus\_creator}: BLOCK\_SYN\_FUN2 \\
\hline
\forall path: PATH;\ mv\_ctxt: \mathbb{F}\ PVALUE;\ alb: A\_LIB\_BLOCK; \\
\quad sd: Z\_SPEC \times (Z\_DECL \times Z\_DECL \times Z\_DECL); \\
\quad used\_mvs: \mathbb{F}\ PVALUE;\ ass: A\_SUBSYS \bullet \\
(path,\ mv\_ctxt,\ alb,\ ass) \mapsto (sd,\ used\_mvs) \in synthesize\_bus\_creator \\
\Leftrightarrow \\
dom\ alb.port\_info.output\_port\_details = \{One\} \\
\land \\
(\exists seqze: seq\ Z\_EXPR;\ pred: Z\_PRED;\ zd: Z\_DECL;\ zs: Z\_SPEC \\
\bullet\ seqze = \\
\qquad \{n,\ w{:}\mathbb{N};\ pv: PVALUE;\ ze: Z\_EXPR; \\
\qquad\quad ipds: PVALUE \mathrel{+\!\!\!\rightarrow} PORT\_DETAILS;\ bpds: seq\ PORT\_DETAILS \\
\qquad \mid pv = num2pvalue\ n \\
\qquad \land\ ipds = alb.port\_info.input\_port\_details \\
\qquad \land\ pv \in dom\ ipds \\
\qquad \land\ ((ipds\ pv).port\_type = ScalarPT \\
\qquad\qquad \land\ ze = ZSequence\langle Ident\ (inport\_name\ pv)\rangle \\
\qquad\quad \lor\ ((ipds\ pv).port\_type = VectorPT\ w\ \lor \\
\qquad\qquad\quad (ipds\ pv).port\_type = BusPT\ (w,\ bpds)) \\
\qquad\qquad \land\ ze = Ident\ (inport\_name\ pv)) \\
\qquad \bullet\ n \mapsto ze\} \\
\land\ pred = PredEq(Application\ (Ident\ (pvalue2ident\ (sc2pv\ \texttt{"}\frown\texttt{/"})), \\
\qquad\qquad\qquad\qquad\qquad\qquad ZSequence\ seqze), \\
\qquad\qquad \{Ident\ (outport\_name\ One)\}) \\
\land\ (zs,\ zd) = wrap\_predicate\ (path,\ \{\})\ (port\_declarations\ alb.port\_info,\ pred) \\
\land\ sd = (zs,\ (zd,\ \langle\rangle,\ \langle\rangle)) \\
\land\ used\_mvs = \{\})
\end{array}$$

For present purposes a *Mux* is the same as a *BusCreator*.

z

$$\begin{array}{l}
\boldsymbol{synthesize\_mux}: BLOCK\_SYN\_FUN2 \\
\hline
synthesize\_mux = synthesize\_bus\_creator
\end{array}$$

6.8.4 Synthesize Bus Selector

Computation of selection position uses the function *selection_position* which is now shared with the specification for virtualization.

Now we give the details for synthesis of *BusSelector* blocks.

z

$$
\begin{array}{l}
\boldsymbol{synthesize\_bus\_selector}: BLOCK\_SYN\_FUN2 \\
\hline
\forall path: PATH;\ mv\_ctxt: \mathbb{P}\ PVALUE;\ alb: A\_LIB\_BLOCK;\ ass: A\_SUBSYS; \\
\quad sd: Z\_SPEC \times (Z\_DECL \times Z\_DECL \times Z\_DECL);\ used\_mvs: \mathbb{F}\ PVALUE \bullet \\
(path,\ mv\_ctxt,\ alb,\ ass) \mapsto (sd,\ used\_mvs) \in synthesize\_bus\_selector \\
\Leftrightarrow \\
dom\ alb.port\_info.input\_port\_details = \{One\} \\
\land \\
(\exists\ par: alb.block\_info.pars;\ mopar: PARAM;\ osp: OUTPUTSIGNALS\_PARAM; \\
\qquad seqze: seq\ Z\_EXPR;\ pred: Z\_PRED;\ pt2: PORT\_TYPE; \\
\qquad selector: (Z\_EXPR \times PORT\_TYPE) \rightarrow seq\ PVALUE \nrightarrow Z\_EXPR; \\
\qquad zs: Z\_SPEC;\ zd: Z\_DECL \\
\bullet\ pt2 = (alb.port\_info.input\_port\_details\ One).port\_type \\
\land\ par.name = OutputSignals \\
\land\ par.value \mapsto osp \in parse\_outputsignals\_param \\
\land\ mopar = (name \mathrel{\hat{=}} sc2pn\ \texttt{"}MuxedOutput\texttt{"},\ value \mathrel{\hat{=}} on) \\
\land\ selector = \quad if\ mopar \in alb.block\_info.pars \\
\qquad\qquad\qquad then\ mux\_selection\ else\ nonmux\_selection \\
\land\ seqze = osp \fatsemi (selector\ (Ident\ (inport\_name\ One),\ pt2)) \\
\land\ pred = \\
\qquad if\ mopar \in alb.block\_info.pars \\
\qquad then\ PredConj \\
\qquad\qquad \{PredEq(Application\ (Ident\ (pvalue2ident\ (sc2pv\ \texttt{"}\frown/\texttt{"})), \\
\qquad\qquad\qquad\qquad ZSequence\ seqze), \\
\qquad\qquad\quad \{Ident\ (outport\_name\ One)\})\} \\
\qquad else\ PredConj \\
\qquad\qquad \{n{:}\mathbb{N};\ ze: Z\_EXPR \\
\qquad\qquad |\ n \mapsto ze \in seqze \\
\qquad\qquad \bullet\ PredEq(ze,\ \{Ident\ (outport\_name(num2pvalue\ n))\})\} \\
\land\ (zs,\ zd) = wrap\_predicate\ (path,\ \{\})\ (port\_declarations\ alb.port\_info,\ pred) \\
\land\ sd = (zs,\ (zd,\ \langle\rangle,\ \langle\rangle)) \\
\land\ used\_mvs = \{\})
\end{array}
$$

6.8.5 Synthesize Demux

The specification of the sequence of selections for the outputs of a *Demux* block is shared in part with virtualization.

The following specifies how to obtain demux selection information from the port map.

z

$$\begin{array}{l}
\boldsymbol{demux\_selection\_positions}: (PVALUE \twoheadrightarrow\!\!\!\!\!+ \; PORT\_DETAILS) \nrightarrow seq\ (\mathbb{N} \times \mathbb{N}) \\
\hline
\forall\ pds: PVALUE \twoheadrightarrow\!\!\!\!\!+ \; PORT\_DETAILS;\ spos: seq\ (\mathbb{N} \times \mathbb{N}) \bullet \\
\quad pds \mapsto spos \in demux\_selection\_positions \\
\Leftrightarrow \\
\quad (\exists\ spds: seq\ PORT\_DETAILS;\ w: \mathbb{N} \\
\quad \bullet\ pds \mapsto spds \in numeric\_port\_sequence \\
\quad \land\ spds \mapsto (spos,\ w) \in demux\_selection\_pos)
\end{array}$$

The following specification of *DeMux* synthesis expects that the bus analysis phase has assigned types to the output ports of block, and will otherwise fail. For the *DeMux* block this information is sufficient to synthesize the block, there is no need to refer to the input port details or the *Outputs* parameter, or the *BusSelectionMode*.

z

$$\begin{array}{l}
\boldsymbol{synthesize\_demux}: BLOCK\_SYN\_FUN2 \\
\hline
\quad \forall path: PATH;\ mv\_ctxt: \mathbb{F}\ PVALUE;\ alb: A\_LIB\_BLOCK;\ ass: A\_SUBSYS; \\
\quad\ sd: Z\_SPEC \times (Z\_DECL \times Z\_DECL \times Z\_DECL);\ used\_mvs: \mathbb{F}\ PVALUE \bullet \\
\quad (path,\ mv\_ctxt,\ alb,\ ass) \mapsto (sd,\ used\_mvs) \in synthesize\_demux \\
\quad \Leftrightarrow \\
\quad dom\ alb.port\_info.input\_port\_details = \{One\} \\
\quad \land \\
\quad (\exists\ opds: PVALUE \twoheadrightarrow\!\!\!\!\!+ \; PORT\_DETAILS;\ ssd: seq\ \mathbb{N} \times \mathbb{N}; \\
\qquad\quad seqze: seq\ Z\_EXPR;\ pred: Z\_PRED;\ pt2: PORT\_TYPE; \\
\qquad\quad zs: Z\_SPEC;\ zd: Z\_DECL \\
\quad \bullet\ opds = alb.port\_info.output\_port\_details \\
\quad \land\ opds \mapsto ssd \in demux\_selection\_positions \\
\quad \land\ pred = \qquad PredConj\{n: dom\ ssd;\ ze: Z\_EXPR \\
\qquad\qquad\qquad\quad \mid (Ident\ (inport\_name\ One)) \mapsto ze \in selection\_exp\ (ssd\ n) \\
\qquad\qquad\qquad\quad \bullet\ PredEq(ze,\ \{Ident\ (outport\_name(num2pvalue\ n))\})\} \\
\quad \land\ (zs,\ zd) = wrap\_predicate\ (path,\ \{\})\ (port\_declarations\ alb.port\_info,\ pred) \\
\quad \land\ sd = (zs,\ (zd,\ \langle\rangle,\ \langle\rangle)) \\
\quad \land\ used\_mvs = \{\})
\end{array}$$

6.8.6 Synthesize Constant

The following is the specification of *Constant* synthesis. Input ports are ignored (there should be none). There must be exactly one output port, number one, otherwise synthesis will fail. There must be a parameter named “Value” which will be translated and used as the value of the output signal.

z

$$
\begin{array}{|l}
\quad \boldsymbol{synthesize\_constant}: BLOCK\_SYN\_FUN2 \\
\hline
\forall mvartypes: MVARTYPES; path: PATH; mv\_ctxt: \mathbb{F}\ PVALUE; alb: A\_LIB\_BLOCK; \\
\quad ass: A\_SUBSYS; used\_mvs: \mathbb{P}\ PVALUE; \\
\quad sd: Z\_SPEC \times (Z\_DECL \times Z\_DECL \times Z\_DECL) \bullet \\
\quad (path, mv\_ctxt, alb, ass) \mapsto (sd, used\_mvs) \in synthesize\_constant \\
\quad \Leftrightarrow \\
\quad dom\ alb.port\_info.output\_port\_details = \{One\} \\
\quad \land \\
\quad (\exists\ ze: Z\_EXPR; pred: Z\_PRED; par: PARAM; zs: Z\_SPEC; zd: Z\_DECL \\
\quad \bullet\ par \in alb.block\_info.pars \\
\quad \land\ par.name = ValuePN \\
\quad \land\ (TMatch\ ze, used\_mvs) = par\_trans2\ mv\_ctxt\ SVM\ par.value \\
\quad \land\ pred = PredEq(ze, \{Ident\ (outport\_name(One))\}) \\
\quad \land\ (zs, zd) \\
\quad\quad = wrap\_predicate\ (path, used\_mvs)\ (port\_declarations\ alb.port\_info, pred) \\
\quad \land\ sd = (zs, (zd, \langle\rangle, \langle\rangle)))
\end{array}
$$

6.8.7 Synthesize Selector

The following is the specification of *Selector* synthesis. There must be exactly one input and one output port, otherwise synthesis will fail. There must be a parameter named “Elements” which will be translated and used in the value of the output signal.

z

$$
\begin{array}{|l}
\quad \boldsymbol{synthesize\_selector}: MVARTYPES \to BLOCK\_SYN\_FUN2 \\
\hline
\forall mvartypes: MVARTYPES;\ path: PATH;\ mv\_ctxt: \mathbb{F}\ PVALUE; \\
\quad alb: A\_LIB\_BLOCK;\ ass: A\_SUBSYS;\ used\_mvs: \mathbb{P}\ PVALUE; \\
\quad sd: Z\_SPEC \times (Z\_DECL \times Z\_DECL \times Z\_DECL) \bullet \\
\quad (path, mv\_ctxt, alb, ass) \mapsto (sd, used\_mvs) \in synthesize\_selector\ mvartypes \\
\quad \Leftrightarrow \\
\quad\quad dom\ alb.port\_info.input\_port\_details = \{One\} \\
\quad \land\ dom\ alb.port\_info.output\_port\_details = \{One\} \\
\quad \land \\
\quad (\exists\ opt: PORT\_TYPE;\ sr: SPECIAL\_RESULT;\ ze: Z\_EXPR; \\
\quad\quad pred: Z\_PRED;\ par: PARAM;\ ln: PVALUE;\ zs: Z\_SPEC;\ zd: Z\_DECL \\
\quad \bullet\ One \mapsto (line\_name \mathrel{\widehat{=}} ln,\ port\_type \mathrel{\widehat{=}} opt) \\
\quad\quad\quad \in alb.port\_info.output\_port\_details \\
\quad \land\ par \in alb.block\_info.pars \\
\quad \land\ par.name = Elements \\
\quad \land\ (sr, used\_mvs) = special\_par\_trans\ mvartypes\ mv\_ctxt\ opt\ par.value \\
\quad \land\ (\ sr = SRVector\ ze \\
\quad\quad\quad \land\ pred = PredEq(ZInfixOps\ (Ident\ (inport\_name(One)), \\
\quad\quad\quad\quad\quad \langle (Composei,\ Ident\ R2zi), \\
\quad\quad\quad\quad\quad (Composei,\ ZBrackets\ ze)\rangle), \\
\quad\quad\quad\quad \{Ident\ (outport\_name(One))\}) \\
\quad\quad \lor\ sr = SRScalar\ ze \\
\quad\quad\quad \land\ pred = PredEq(Application\ (Ident\ (inport\_name(One)), \\
\quad\quad\quad\quad\quad ZBrackets(Application(Ident\ R2zi,\ ze))), \\
\quad\quad\quad\quad \{Ident\ (outport\_name(One))\})) \\
\quad \land\ (zs, zd) \\
\quad\quad = wrap\_predicate\ (path, used\_mvs)\ (port\_declarations\ alb.port\_info, pred) \\
\quad \land\ sd = (zs, (zd, \langle\rangle, \langle\rangle)))
\end{array}
$$

6.8.8 Synthesize Merge

The following is the specification of *Merge* synthesis.

There must be exactly one output port, otherwise synthesis will fail.

If the action complex is “open” then a schema with internal state will be synthesized. In that case there must be a parameter named “InitialOutput” which will be translated and used as the initial value of the state.

Whether or not there is state the synthesized block will have additional “Action” input ports, one for each inport, and these determine which value is selected for the output. If the schema has state and none of them is selected then the value of the state variable is output.

First we define a functions which construct the declaration part of the schemas.

z

$$\boldsymbol{closed\_merge\_invocations}: A\_LIB\_BLOCK \rightarrow \mathbb{F}\ INVOCATION$$

$$\forall alb: A\_LIB\_BLOCK;\ m: \mathbb{N}$$
$$\mid (name \mathrel{\widehat=} Inputs,\ value \mathrel{\widehat=} num2pvalue\ m) \in alb.block\_info.pars$$
$$\bullet \quad closed\_merge\_invocations\ alb$$
$$= port\_invocations\ (alb.port\_info) \cup action\_port\_invocations\ m$$

z

$$\boldsymbol{open\_merge\_invocations}: A\_LIB\_BLOCK \rightarrow \mathbb{F}\ INVOCATION$$

$$\forall alb: A\_LIB\_BLOCK \bullet$$
$$open\_merge\_invocations\ alb$$
$$= closed\_merge\_invocations\ alb$$
$$\cup\ state\_invocation\ (\!|\{state,\ stateP,\ initial\_state\}|\!)$$

z

$$\boldsymbol{closed\_merge\_decl}: A\_LIB\_BLOCK \rightarrow Z\_DECL$$

$$\forall alb: A\_LIB\_BLOCK \bullet$$
$$closed\_merge\_decl\ alb$$
$$= zdecl\_from\_invs\ (closed\_merge\_invocations\ alb)$$

z

$$\boldsymbol{open\_merge\_decl}: A\_LIB\_BLOCK \rightarrow Z\_DECL$$

$$\forall alb: A\_LIB\_BLOCK \bullet$$
$$closed\_merge\_decl\ alb$$
$$= zdecl\_from\_invs\ (open\_merge\_invocations\ alb)$$

Then functions which construct the predicate.

z

$$\begin{array}{l}
\textbf{\textit{closed\_merge\_pred}}: \mathbb{N}_1 \rightarrow Z\_PRED \\
\hline
\forall n: \mathbb{N}_1 \bullet \\
\quad closed\_merge\_pred\ n \\
\quad = PredDisj \\
\qquad \{m: \mathbb{N}_1 \\
\qquad \mid m \leq n \\
\qquad \bullet\ PredConj\ \{ \\
\qquad\quad PredBool\ (Ident\ (action\_port\_id\ m)), \\
\qquad\quad PredEq\ (\quad Ident\ (inport\_name\ (num2pvalue\ m)), \\
\qquad\qquad \{Ident\ (outport\_name\ One)\})\} \\
\qquad \}
\end{array}$$

In the following, absence of the *InitialOutput* parameter, or its having the value "[]", should cause an *Error* to be raised (not a warning). Synthesis of the *Merge* should fail, but processing should continue.

The PATH parameter in the following is for error reporting only.

z

$$\begin{array}{l}
\textbf{\textit{open\_merge\_pred}}: (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ PARAM) \times PATH \\
\qquad \nrightarrow Z\_PRED \times \mathbb{F}\ PVALUE \\
\hline
\forall mv\_ctxt: \mathbb{F}\ PVALUE;\ pars: \mathbb{F}\ PARAM;\ path: PATH; \\
\quad pred: Z\_PRED;\ used\_mvs: \mathbb{F}\ PVALUE \bullet \\
\quad (mv\_ctxt, pars, path) \mapsto (pred, used\_mvs) \in open\_merge\_pred \\
\quad \Leftrightarrow \\
\quad (\exists n: \mathbb{N}_1;\ par: pars;\ ze: Z\_EXPR \\
\quad \bullet\ (name \mathrel{\widehat{=}} Inputs, value \mathrel{\widehat{=}} num2pvalue\ n) \in pars \\
\quad \land\ par.name = InitialOutput \\
\quad \land\ \neg\ par.value = sc2pv\ \texttt{"[]"} \\
\quad \land\ (TMatch\ ze, used\_mvs) = par\_trans2\ mv\_ctxt\ SVM\ par.value \\
\quad \land\ pred = PredConj\ \{closed\_merge\_pred\ n, \\
\qquad PredEq\ (Ident\ (outport\_name\ One), \{Ident\ (pvalue2ident\ stateP)\}), \\
\qquad PredEq\ (ze, \{Ident\ (pvalue2ident\ initial\_state)\})\} \\
\quad)
\end{array}$$

Then a function for each of the hold and reset schemas, and one which generates whichever of these is required according to the hold context.

Then these are combined to give the overall merge synthesis specification.

z

$$
\begin{array}{l}
\quad \boldsymbol{synthesize\_closed\_merge}: MVARTYPES \to BLOCK\_SYN\_FUN2 \\
\hline
\forall mvartypes: MVARTYPES; path: PATH; mv\_ctxt: \mathbb{F}\ PVALUE; alb: A\_LIB\_BLOCK; \\
\quad ass: A\_SUBSYS; used\_mvs: \mathbb{P}\ PVALUE; \\
\quad sd: Z\_SPEC \times (Z\_DECL \times Z\_DECL \times Z\_DECL) \bullet \\
\quad (path, mv\_ctxt, alb, ass) \mapsto (sd, used\_mvs) \in synthesize\_closed\_merge\ mvartypes \\
\quad \Leftrightarrow \\
\quad (\exists\ par: PARAM; n: \mathbb{N}_1; pred: Z\_PRED; decl: Z\_DECL; \\
\quad\quad zs: Z\_SPEC; zd: Z\_DECL \\
\quad \bullet (name \mathrel{\widehat{=}} Inputs, value \mathrel{\widehat{=}} num2pvalue\ n) \in alb.block\_info.pars \\
\quad \land pred = closed\_merge\_pred\ n \\
\quad \land decl = closed\_merge\_decl\ alb \\
\quad \land used\_mvs = \{\} \\
\quad \land (zs, zd) = wrap\_predicate\ (path, used\_mvs)\ (decl, pred) \\
\quad \land sd = (zs, (zd, \langle\rangle, \langle\rangle)))
\end{array}
$$

z

$$
\begin{array}{l}
\quad \boldsymbol{synthesize\_open\_merge}: MVARTYPES \to BLOCK\_SYN\_FUN2 \\
\hline
\forall mvartypes: MVARTYPES; path: PATH; mv\_ctxt: \mathbb{F}\ PVALUE; \\
\ alb: A\_LIB\_BLOCK; ass: A\_SUBSYS; used\_mvs: \mathbb{P}\ PVALUE; \\
\ sd: Z\_SPEC \times (Z\_DECL \times Z\_DECL \times Z\_DECL) \bullet \\
\quad (path, mv\_ctxt, alb, ass) \mapsto (sd, used\_mvs) \in synthesize\_open\_merge\ mvartypes \\
\quad \Leftrightarrow \\
\quad (\exists\ pred: Z\_PRED; decl: Z\_DECL \\
\quad \bullet (mv\_ctxt, alb.block\_info.pars, path) \mapsto (pred, used\_mvs) \in open\_merge\_pred \\
\quad \land decl = open\_merge\_decl\ alb \\
\quad \land sd = state\_wrap\_predicate\ (path, used\_mvs, decl, pred, \\
\quad\quad\quad ass.subsys\_info.action\_info.held\_context, \{(state, stateP, initial\_state)\}))
\end{array}
$$

z

$$
\begin{array}{l}
\quad \boldsymbol{open\_merge}: \mathbb{P}\ (PATH \times A\_SUBSYS) \\
\hline
\forall path: PATH; ass: A\_SUBSYS \bullet \\
\quad (path, ass) \in open\_merge \\
\quad \Leftrightarrow (\exists\ ac: ACTION\_COMPLEX \bullet ac.tail = last\ path \\
\quad\quad\quad \land ac \in ass.subsys\_info.action\_info.complexes \\
\quad\quad\quad \land ac.open)
\end{array}
$$

z

$$
\begin{array}{|l}
\quad \boldsymbol{synthesize\_merge}: MVARTYPES \to BLOCK\_SYN\_FUN2 \\
\hline
\forall mvartypes: MVARTYPES; path: PATH; mv\_ctxt: \mathbb{F}\ PVALUE; alb: A\_LIB\_BLOCK; \\
\quad ass: A\_SUBSYS; used\_mvs: \mathbb{P}\ PVALUE; \\
\quad sd: Z\_SPEC \times (Z\_DECL \times Z\_DECL \times Z\_DECL) \bullet \\
\quad (path, mv\_ctxt, alb, ass) \mapsto (sd, used\_mvs) \in synthesize\_merge\ mvartypes \\
\quad \Leftrightarrow \\
\quad\quad dom\ alb.port\_info.output\_port\_details = \{One\} \\
\quad \land \neg\ (name \mathrel{\widehat{=}} AllowUnequalInputPortWidths, value \mathrel{\widehat{=}} on) \in alb.block\_info.pars \\
\quad \land ((path, ass) \in open\_merge \\
\quad\quad \land (path, mv\_ctxt, alb, ass) \\
\quad\quad\quad \mapsto (sd, used\_mvs) \in synthesize\_open\_merge\ mvartypes) \\
\quad\quad \lor \\
\quad\quad (\neg\ (path, ass) \in open\_merge \\
\quad\quad \land (path, mv\_ctxt, alb, ass) \\
\quad\quad\quad \mapsto (sd, used\_mvs) \in synthesize\_closed\_merge\ mvartypes)
\end{array}
$$

6.8.9 Synthesize Terminator

The following is the specification of *Terminator* synthesis. Input ports are ignored.

z

$$
\begin{array}{|l}
\quad \boldsymbol{synthesize\_terminator}: BLOCK\_SYN\_FUN2 \\
\hline
\quad \forall path: PATH; mv\_ctxt: \mathbb{F}\ PVALUE; alb: A\_LIB\_BLOCK; ass: A\_SUBSYS; \\
\quad\quad sd: Z\_SPEC \times (Z\_DECL \times Z\_DECL \times Z\_DECL); used\_mvs: \mathbb{P}\ PVALUE \bullet \\
\quad (path, mv\_ctxt, alb, ass) \mapsto (sd, used\_mvs) \in synthesize\_terminator \\
\quad \Leftrightarrow \\
\quad (\exists\ zs: Z\_SPEC; zd: Z\_DECL \\
\quad \bullet (zs, zd) \\
\quad\quad = wrap\_predicate\ (path, \{\})\ (port\_declarations\ alb.port\_info, PredConj\{\}) \\
\quad \land sd = (zs, (zd, \langle\rangle, \langle\rangle)) \\
\quad \land used\_mvs = \{\})
\end{array}
$$

6.8.10 The Synthesis Traversal

The following table shows the correspondence between the block types for which a synthesis method has been specified and the method for the block.

z

$$\boldsymbol{block\_syn\_table}: MVARTYPES \rightarrow PVALUE \mathrel{\ooalign{$\rightarrow$\cr\hfil$\shortmid\shortmid$\hfil}} BLOCK\_SYN\_FUN$$

$$\begin{array}{lll}
\forall mvartypes: MVARTYPES \bullet\ block\_syn\_table\ mvartypes = \{ \\
BusCreator & \mapsto lift\_bsf\ synthesize\_bus\_creator, \\
BusSelector & \mapsto lift\_bsf\ synthesize\_bus\_selector, \\
Constant & \mapsto lift\_bsf\ synthesize\_constant, \\
Demux & \mapsto lift\_bsf\ synthesize\_demux, \\
Merge & \mapsto lift\_bsf\ (synthesize\_merge\ mvartypes), \\
Mux & \mapsto lift\_bsf\ synthesize\_mux, \\
Selector & \mapsto lift\_bsf\ (synthesize\_selector\ mvartypes), \\
Terminator & \mapsto lift\_bsf\ synthesize\_terminator \\
\}
\end{array}$$

Before arranging for this to be mapped over the *A_BLOCK*, we need a spec of a *BLOCK_SYN_FUN* which will create a dummy invocation for use in the output where instantiation and synthesis has failed.

z

$$\boldsymbol{dummy\_blocksyn}: PATH \rightarrow A\_LIB\_BLOCK \rightarrow A\_LIB\_BLOCK$$

$$\begin{array}{l}
\forall path: PATH;\ alb,\ alb': A\_LIB\_BLOCK \bullet \\
(alb,\ alb') \in dummy\_blocksyn\ path \\
\Leftrightarrow \\
alb' = \\
(block\_info \mathrel{\widehat=} \\
\quad (pars \mathrel{\widehat=} alb.block\_info.pars, \\
\quad input\_port\_types \mathrel{\widehat=} alb.block\_info.input\_port\_types, \\
\quad output\_port\_types \mathrel{\widehat=} alb.block\_info.output\_port\_types, \\
\quad specification \mathrel{\widehat=} \langle\rangle, \\
\quad invocation \mathrel{\widehat=} (OtherInv\ (path2loci\ path), \\
\qquad\qquad (\langle DecDec\ (\\
\qquad\qquad\qquad names \mathrel{\widehat=} \langle path2loci\ path\rangle, \\
\qquad\qquad\qquad type \mathrel{\widehat=} Ident\ Ui)\rangle,\ \langle\rangle,\ \langle\rangle)), \\
\quad virtual \mathrel{\widehat=} alb.block\_info.virtual, \\
\quad used\_maskvars \mathrel{\widehat=} \{\}), \\
port\_info \mathrel{\widehat=} alb.port\_info)
\end{array}$$

Now we arrange for this to be mapped over the *A_BLOCK*.

The context for this traversal is the combination of the path and the set of maskvariables, which is obtained from the *mv_ctxt* of the enclosing subsystem. The following function updates this context.

z

$$\begin{array}{l}
\boldsymbol{newc\_blocksyn}: (PATH \times (\mathbb{F}\ PVALUE) \times A\_SUBSYS) \rightarrow A\_SUBSYS \\
\qquad \rightarrow PVALUE \rightarrow (PATH \times (\mathbb{F}\ PVALUE) \times A\_SUBSYS) \\
\hline
\forall\ path,\ path': PATH;\ maskvars,\ newmaskvars: \mathbb{F}\ PVALUE; \\
\qquad ass,\ asso: A\_SUBSYS;\ pv: PVALUE \\
\mid path' = path \frown \langle pv \rangle \\
\wedge\ newmaskvars = ass.subsys\_info.mv\_ctxt \\
\bullet\ newc\_blocksyn\ (path,\ maskvars,\ asso)\ ass\ pv \\
\qquad = (path',\ newmaskvars,\ ass)
\end{array}$$

In the following, the path set returned is empty unless the invocation is empty after any attempted synthesis. I.e. we test the invocation before synthesis to see whether to attempt synthesis, and we test it after synthesis to check whether it succeeded. Note that there are no synthesis functions for port block types, and the implementation may assume this.

z

$$
\begin{array}{l}
\mathbf{libmap\_blocksyn}: MVARTYPES \rightarrow PATH \times (\mathbb{F}\ PVALUE) \times A\_SUBSYS \\
\qquad \rightarrow A\_LIB\_BLOCK \rightarrow (\mathbb{F}\ PATH) \times A\_LIB\_BLOCK \\
\hline
\quad \forall\ mvartypes: MVARTYPES;\ path: PATH;\ mv\_ctxt: \mathbb{F}\ PVALUE; \\
\qquad alb: A\_LIB\_BLOCK;\ ass: A\_SUBSYS \bullet \\
\quad libmap\_blocksyn\ mvartypes\ (path,\ mv\_ctxt,\ ass)\ alb = \\
\quad (\mu\ alb': A\_LIB\_BLOCK;\ param: PARAM;\ bt: PVALUE; \\
\qquad bsf: BLOCK\_SYN\_FUN;\ paths: \mathbb{F}\ PATH \\
\quad |\ param \in alb.block\_info.pars \\
\quad \land\ param = (name \mathrel{\widehat{=}} BlockType,\ value \mathrel{\widehat{=}} bt) \\
\quad \land\ (alb.block\_info.invocation = (NoInv,\ (\langle\rangle,\ \langle\rangle,\ \langle\rangle)) \\
\quad\ \ \land\ (alb.block\_info.virtual = VUnknown \\
\quad\ \ \ \ \lor\ alb.block\_info.virtual = VInhibit) \\
\quad\ \ \land\ (\quad bt \mapsto bsf \in block\_syn\_table\ mvartypes \\
\qquad\quad \land\ (ALibBlock\ alb) \mapsto (ALibBlock\ alb') \in bsf\ (path,\ mv\_ctxt,\ ass) \\
\qquad\quad \land\ paths = \{\} \\
\quad\ \ \ \ \lor\quad (bt \notin (dom\ (block\_syn\_table\ mvartypes) \cup port\_block\_types) \\
\qquad\qquad\quad \lor\ bt \mapsto bsf \in block\_syn\_table\ mvartypes \\
\qquad\qquad\quad\ \ \land\ (ALibBlock\ alb) \notin dom\ (bsf\ (path,\ mv\_ctxt,\ ass))) \\
\qquad\quad \land\ alb' = dummy\_blocksyn\ path\ alb \\
\qquad\quad \land\ paths = \{path\} \\
\quad\ \ \ ) \\
\quad\ \ \lor\ (alb.block\_info.invocation \neq (NoInv,\ (\langle\rangle,\ \langle\rangle,\ \langle\rangle)) \\
\quad\ \ \ \ \lor\ bt \in port\_block\_types \\
\quad\ \ \ \ \lor\ (\exists emap: PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} Z\_EXPR \bullet alb.block\_info.virtual = Virtual\ emap)) \\
\qquad\quad \land\ alb' = alb \land paths = \{\}) \\
\quad \bullet \qquad (paths,\ alb'))
\end{array}
$$

The processing of subsystems simply aggregates the set of failure paths:

z

$$
\begin{array}{l}
\mathbf{ssmap\_blocksyn}: PATH \times (\mathbb{F}\ PVALUE) \times A\_SUBSYS \\
\qquad \rightarrow A\_SUBSYS \times (PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} \mathbb{F}\ PATH) \\
\qquad \rightarrow (\mathbb{F}\ PATH) \times A\_SUBSYS \\
\hline
\quad \forall\ path: PATH;\ mv\_ctxt: \mathbb{F}\ PVALUE;\ eass,\ ass: A\_SUBSYS; \\
\quad\ \ rmap: PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} \mathbb{F}\ PATH \bullet \\
\quad ssmap\_blocksyn\ (path,\ mv\_ctxt,\ eass)\ (ass,\ rmap) = (\bigcup\ (ran\ rmap),\ ass)
\end{array}
$$

We then map this function over the *A_BLOCK*.

The result is an updated *A_BLOCK* and a set of paths of untranslated blocks.

Note that the top-level subsystem is passed itself as its enclosing subsystem, perhaps a dummy would be better. It is not intended that this value be used, it is only required when processing library blocks.

z

$$\begin{array}{l}
\boldsymbol{synthesize\_blocks}: MVARTYPES \to A\_BLOCK \to (\mathbb{F}\ PATH) \times A\_BLOCK \\
\hline
\forall\ mvartypes: MVARTYPES;\ ab: A\_BLOCK;\ ass: A\_SUBSYS \\
\mid ab = ASubsys\ ass \\
\bullet\ synthesize\_blocks\ mvartypes\ ab = a\_block\_map\_cr \\
\quad (libmap\_blocksyn\ mvartypes,\ ssmap\_blocksyn,\ newc\_blocksyn) \\
\quad (\langle param\_value\ ass.subsys\_info.syspars\ Name \rangle, \\
\quad\ ass.subsys\_info.mv\_ctxt, \\
\quad\ ass) \\
\quad ab
\end{array}$$

6.9 Translating Subsystems

The specifications in this section describe generation of the specification and invocation of the subsystems of the model and the top level system.

6.9.1 General Description

In order to support action subsystems it is necessary not only to have specifications of how a subsystem behaves in normal operation, but also to have specifications for the behaviour when an action subsystem is not active. This requires similar specifications for the subsystems or library blocks which occur at any depth inside an action subsystem, and also, when compiling library blocks, for all blocks irrespective of whether they occur in an action subsystem in the library (in case they end up being referred to from an action subsystem in a model using the library).

Two distinct effects may be required. The first is to hold the state of a subsystem or library block, the second is to reset it. Which is required depends upon a parameter on the action block in the smallest enclosing action subsystem (if there is one, otherwise neither are needed in a model but both are needed in a library).

The following description is adapted from that in the proposal for support of action subsystems [3].

Subsystems containing action subsystems differ from other subsystems only in having additional equations passing action port values to *Merge* blocks. The additional schemas are used directly or indirectly in the definition of action subsystems, and may also be used in defining corresponding schemas at higher levels in the subsystem hierarchy.

The schemas generated for a block depend upon the context in which the block occurs. For this purpose the type *HOLD_CONTEXT* has been defined.

Names are to be given to the new schemas using suffixing conventions which are specified using the functions i_a, i_h, i_r, w_a, w_h and w_r (applying the required suffixes for active, held and reset schemas to identifiers and words respectively).

The following is an informal sketch of the Z which is generated.

1. For each block which has some internal state and whose held context is *held* or *unknown*:

   an extra schema, the *block state held* schema. This schema equates the before and after values for every component of the state of the block. It is named using suffix $_h$.

2. For each block which has some internal state and whose held context is *reset* or *unknown*:

   an extra schema, the *block state reset* schema. This schema equates the after values for each component of the state of the block with the initial value of that state component (if specified) and otherwise with real zero. It is named using suffix $_r$.

3. Block state held and block state reset schemas for manually specified library blocks are expected to be in the Z library with additional metadata giving the names of these schemas. Where a library block is parameterized the state held and reset schemas are expected to be identically parameterised.

4. Block state held schemas for subsystems consist of signature only. The components are all those of the corresponding subsystem. The *type* in the signature is the name of block state held or reset schema for the relevant block. If only one of these is available then that one is used, if both are available the held schema is used. If no held or reset schema is available then the type is set to $\mathbb{U}$.

   e.g. $[b1 : B1_h; b2 : B2_h...]$

5. Block state reset schemas for subsystems consist of signature only. The components are all those of the corresponding subsystem. The *type* in the signature is the name of block state held or reset schema for the relevant block. If only one of these is available then that one is used, if both are available the reset schema is used. If no held or reset schema is available then the type is set to $\mathbb{U}$.

   e.g. $[b1 : B1_r; b2 : B2_r...]$

6. Block state held and block state reset schemas for synthesized blocks with internal state must be generated by the synthesis code.

7. Schemas for normal subsystems which contain action subsystems are augmented by the addition of further line equations which connect the action signals controlling action subsystems to extra ports supplied for this purpose on the merge blocks to which the action subsystems are connected.

8. Multiple schemas will be generated to define each action subsystem as follows:

   (a) block state held and/or block state reset schemas are generated as specified above (depending on the held context).

   (b) a *block active* schema is generated which is the similar to the schema for an ordinary (not action) subsystem except that:

- the *Action Port* block is given special treatment analogous to that of other ports and results in a single item "Action?:$\mathbb{U}$" in the signature.

The block active schema is named with suffix $_a$.

(c) an overall block schema (with standard subsystem name as at present) is generated with the following form (as a pidgin Z schema expression):

(i) if the action subsystem has no state:
$B = Active \wedge B_a \vee Inactive$

(ii) if the action block specifies that the state should be held:
$B = Active \wedge B_a \vee Inactive \wedge B_h$

(iii) if the action block specifies that the state should be reset:
$B = Active \wedge B_a \vee Inactive \wedge B_r$

Where B is the full Z name for the subsystem.

6.9.2 Data Types

In order to support action subsystems it has been convenient to make many of the specifications sensitive to the kind of Z paragraph which is being produced. The "declaration types" are:

DTActive for the active schema for an action subsystem

DTHeld for a state held schema

DTReset for a state reset schema

DTPlain for a normal subsystem schema

```
z
DTYPE ::=
          DTActive
      |   DTHeld
      |   DTReset
      |   DTPlain
```

6.9.3 The Declaration Part

This specification determines the declaration part or signature of the schemas generated for a subsystem, and covers not only the schema for an ordinary subsystem but also the active schema for an action subsystem and held and reset schemas as required. For this purpose it is passed a *DTYPE* which indicates what kind of schema is required. The signature entry for any block is selected according to the value of the *DTYPE* parameter, taking into account the availability of held and reset schemas. A check is made on whether any held or reset schemas are referred to, and the result of this check is returned as a boolean which allows generation of trivial held and reset schemas to be inhibited.

The declaration part of the box contains one variable declaration for each block in the system diagram except for blocks which have been virtualized.

For each input or output port a simple declaration assigning an unknown type $\mathbb{U}$ will be included. This will be resolved by type inference in the ProofPower-Z type checker. Action, trigger and enable blocks are included in the declarations.

The schema declaration list is formed by sorting all the invocations, and then extracting and concatenating the declarations from them.

This function is now parameterized by a *DTYPE* which is used to select the invocation to be used in the declaration, permitting the function to be used for held and reset schemas. The method used here is to select an appropriate declaration from the invocation (which contains three declarations for use in different contexts) and then sort and concatenate the declarations. Where no declaration is available $\mathbb{U}$ is used, but it is presumed that this will only happen for declations whose key is *OtherInv*, i.e. subsystems or library blocks.

z

```
schemadecs: DTYPE → (PVALUE ⇸ A_BLOCK) → (Z_DECL × BOOL)
─────────────────────────
∀ dtype: DTYPE; blocks: PVALUE ⇸ A_BLOCK; empty: BOOL;
  inv_map, oi_map: PVALUE ⇸ INVOCATION;
  oi_map2, i_map: PVALUE ⇸ INVKEY × Z_DECL
| inv_map = blocks ⨾ get_invocation
∧ oi_map = inv_map ▷ {i: IDENT;  is: 𝕌 • (OtherInv i, is)}
∧ oi_map2 = oi_map ⨾
        {i: INVKEY;  is1, is2, is3, is1′: 𝕌
        | is1′ =        if dtype = DTHeld
                        then   if is2 = ⟨⟩
                               then is3
                               else is2
                        else   if dtype = DTReset
                               then   if is3 = ⟨⟩
                                      then is2
                                      else is3
                               else is1
        ∧ is1′ ≠ ⟨⟩
        • (i, (is1, is2, is3)) ↦ (i, is1′)}
∧ empty = (oi_map2 = {})
∧ i_map = (oi_map ⨾
                {i: IDENT;  is: 𝕌; is1: Z_DEC
                | is1 = DecDec (names ≘ ⟨i⟩, type ≘ Ident Ui)
                • (OtherInv i, is) ↦ (OtherInv i, ⟨is1⟩)
                })
          ⊕ oi_map2
          ∪ ((inv_map \ oi_map)
            ⨾   {i: INVKEY;  is: 𝕌
                • (i, is) ↦ (i, main_inv is)})
• schemadecs dtype blocks =
(⁀/ (sort_by_invkey (ran i_map) ⨾ second), empty)
```

6.9.4 The Specification

The following functions creates the Z specifications for subsystems.

There are two versions, the first is used for all those specifications which are given as schemas (including abbreviation definitions with abstractions on the right containing horizontal schema expressions). This is used for non-action subsystems, for state held and reset schemas, and for the active schema for an action subsystem. The second is for the top level specification of an action subsystem, which is always an abbreviation definition on the right of which is a schema expression, possible embedded in an abstraction. In both cases an abstraction is used if the set of variables supplied (which should be the union of the set of maskvariables on this subsystem and the set of mask variables from higher level subsystems which are used in this subsystem) is non-empty.

For the first case if there are no mask variables on the subsystem and no local variables used in it, then the specification will be a schema box. Otherwise an abbreviation definition is used with the corresponding horizontal schema enclosed in an abstraction.

z

$$
\begin{array}{|l}
\boldsymbol{make\_ss\_spec}: WORD \rightarrow Z\_DECL \rightarrow Z\_PRED \\
\qquad \rightarrow \mathbb{F}\ PVALUE \rightarrow seq\ Z\_PARA \\
\hline
\forall\ name: WORD;\ decl: Z\_DECL;\ pred: Z\_PRED;\ vars: \mathbb{F}\ PVALUE \bullet \\
make\_ss\_spec\ name\ decl\ pred\ vars \\
= \\
if\ vars = \{\} \\
then\ \langle SchemaBox\ (\theta Z\_SCHEMABOX)\rangle \\
else\quad (\mu Z\_ABBREVDEF \\
\qquad \mid ident = word2ident\ name \\
\qquad \land value = ZLambdaExp\ (\\
\qquad\qquad decl \mathrel{\widehat{=}} \langle DecSchema(vars2uhschem\ vars)\rangle, \\
\qquad\qquad exp \mathrel{\widehat{=}} ZHSchema\ (\theta Z\_HSCHEMA)) \\
\qquad \bullet \langle AbbrevDef\ (\theta Z\_ABBREVDEF)\rangle)
\end{array}
$$

The translation of the definition for an action subsystem is done using a schema expression involving an active schema and possible a held or reset schema (as well as the schemas *Active* and *Inactive* which test the value on the *Action?* port). The expression used to invoke these schema is derived from the invocation field for the relevant schema, taking the declaration part of that schema.

z

$$
\begin{array}{|l}
\boldsymbol{type\_from\_decl}: Z\_DECL \rightarrowtail\!\!\!\!\!\!+ Z\_EXPR \\
\hline
\forall\ decl: Z\_DECL;\ ze: Z\_EXPR \bullet \\
decl \mapsto ze \in type\_from\_decl \\
\Leftrightarrow \\
(\exists\ names: seq\ IDENT \bullet \\
decl = \langle DecDec\ (names \mathrel{\widehat{=}} names,\ type \mathrel{\widehat{=}} ze)\rangle)
\end{array}
$$

Now we specify how to make the specification for an action subsystem.

make_ass_spec expects to be passed:

name the z name of the specification to be produced

adecl the declaration from the invocation for the active schema

sdecl the declaration from the invocation for the state held or reset schema (as appropriate)

vars a (possibly empty) set over variables over which (if non-empty) abstraction will take place

The declarations are used as sources for appropriately instantiated schema references and must be of the form "localname: schemaref" not of the form "localname: $\mathbb{U}$". If the schema requires parameters they must of course be present.

make_ass_spec is formally a total function even though for some values in the domain *type_from_decl* will fail. However, it should never be called with such values and if it is this may be treated as a fatal internal error in ClawZ. As far as the Z is concerned the value returned in these circumstances is arbitrary and irrelevant.

z

```
$\boldsymbol{make\_ass\_spec}: (WORD \times Z\_DECL \times Z\_DECL \times \mathbb{F}\ PVALUE)$
        $\rightarrow seq\ Z\_PARA$
────────────────────────────
$\forall\ name: WORD; adecl, sdecl: Z\_DECL; vars: \mathbb{F}\ PVALUE; szp: seq\ Z\_PARA \bullet$
$(name, adecl, sdecl, vars) \mapsto szp \in make\_ass\_spec$
$\Leftrightarrow$
$(\exists\ as, rhd, body: Z\_EXPR$
$\bullet\ type\_from\_decl\ adecl = as$
$\land\ rhd =$    $if\ sdecl = \langle\rangle$
                  $then\ SchemaRef\ Inactive$
                  $else\ ZSConj\ (SchemaRef\ Inactive, type\_from\_decl\ sdecl)$
$\land\ body = ZSDisj\ (ZSConj\ (SchemaRef\ Active, as), rhd)$
$\land\ szp =$
        $\langle(\mu Z\_ABBREVDEF$
          $|\ ident = word2ident\ name$
          $\land\ value =$    $if\ vars = \{\}$
                              $then\ body$
                              $else\ ZLambdaExp\ ($
                                      $decl \mathrel{\widehat{=}} \langle DecSchema(vars2uhschem\ vars)\rangle,$
                                      $exp \mathrel{\widehat{=}} body)$
          $\bullet\ AbbrevDef\ (\theta Z\_ABBREVDEF))$
        $\rangle)$
```

6.9.5 The Invocation

The actual parameters to a masked subsystem are supplied at the point of invocation. This differs from the parameterisation of library blocks. The reason for this difference is that the block name derived from the path, which is the name used for the definition which invokes a library block will already have been used to define the schema block. If the parameters are supplied outside the enclosing schema then another name will be needed, and its not obvious how to chose a new unique name.

We now specify the binding which is passed as a parameter to a subsystem. This depends upon the parameters of the subsystem (of which the various mask parameters are the ones consulted) and the set of variables which have already been masked in this context (i.e. the *mv_ctxt*). In addition to the information defining the relevant binding two sets of variable names are returned, the set of variables masked by this subsystem, and the set of masked variables (of subsystems higher up) used in the mask parameters.

z

$$
\begin{array}{l}
\boldsymbol{make\_maskbinding}: (\mathbb{F}\ PARAM) \times (\mathbb{F}\ PVALUE) \\
\qquad \rightarrow (\mathbb{F}\ (IDENT \times Z\_EXPR)) \times (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ PVALUE) \\
\hline
make\_maskbinding = (((\mathbb{F}\ PARAM) \times \mathbb{F}\ PVALUE) \times \{(\{\},\{\},\{\})\}) \oplus \\
\{pars: \mathbb{F}\ PARAM;\ binding: \mathbb{F}\ (IDENT \times Z\_EXPR);\ pmvarp: MASK\_VAR\_PAR; \\
\ pmstyp: MASK\_STYLE\_PAR;\ pmvalp: MASK\_VALUE\_PAR; \\
\ mvarp,\ mstyp,\ mvalp: PVALUE; \\
\ mv\_ctxt,\ new\_maskvars,\ used\_maskvars: \mathbb{F}\ PVALUE; \\
\ binding\_info: PVALUE \twoheadrightarrow\!\!\!\!\!\!+\ Z\_EXPR \times (\mathbb{F}\ PVALUE) \\
\mid \{sc2pn\ \texttt{"}MaskVariables\texttt{"} \mapsto mvarp,\ sc2pn\ \texttt{"}MaskStyleString\texttt{"} \mapsto mstyp, \\
\quad sc2pn\ \texttt{"}MaskValueString\texttt{"} \mapsto mvalp\} \subseteq (param\_value\ pars) \\
\land\ mvarp \mapsto pmvarp \in parse\_maskvar\_param \\
\land\ mstyp \mapsto pmstyp \in parse\_maskstyle\_param \\
\land\ mvalp \mapsto pmvalp \in parse\_maskvalue\_param \\
\land\ dom\ pmvarp = dom\ pmstyp = dom\ pmvalp \\
\land\ binding\_info = \\
\qquad \{n: dom\ pmvarp;\ name: PVALUE;\ style: MASK\_STYLE; \\
\qquad\ value: Z\_EXPR;\ umvs: \mathbb{F}\ PVALUE \\
\qquad \mid name = pmvarp\ n \\
\qquad \land\ style = pmstyp\ n \\
\qquad \land\ (TMatch\ value,\ umvs) = maskparam\_trans\ (style,\ pmvalp\ n,\ mv\_ctxt) \\
\qquad \bullet\ name \mapsto (value,\ umvs) \\
\qquad \} \\
\land\ new\_maskvars = dom\ binding\_info \\
\land\ used\_maskvars = \bigcup\ (ran\ (binding\_info \fatsemi second)) \\
\land\ binding = \quad \{pv: PVALUE;\ ze: Z\_EXPR;\ mvs: \mathbb{F}\ PVALUE \\
\qquad\qquad\quad \mid pv \mapsto (ze,\ mvs) \in binding\_info \\
\qquad\qquad\quad \bullet\ pvalue2ident\ pv \mapsto ze\} \\
\bullet\ (pars,\ mv\_ctxt) \mapsto (binding,\ new\_maskvars,\ used\_maskvars)\}
\end{array}
$$

Though different methods are used in the construction of the specification of action subsystems, the construction of the invariant requires no special treatment. The following specification therefore covers all the invocations needed while translating subsystems.

z

$$
\begin{array}{l}
\boldsymbol{make\_ss\_inv}: IDENT \rightarrow WORD \rightarrow \mathbb{F}\ PVALUE \\
\qquad \rightarrow \mathbb{F}\ (IDENT \times Z\_EXPR) \rightarrow Z\_DECL \\
\hline
\forall\ name: IDENT;\ ssname: WORD;\ vars: \mathbb{F}\ PVALUE; \\
\qquad binding,\ binding': \mathbb{F}\ (IDENT \times Z\_EXPR) \\
\mid binding' = \{v{:}vars \bullet pvalue2ident\ v \mapsto Ident\ (pvalue2ident\ v)\} \oplus binding \\
\bullet\ make\_ss\_inv\ name\ ssname\ vars\ binding \\
= \langle DecDec(\\
\qquad names \mathrel{\widehat{=}} \langle name \rangle, \\
\qquad type \mathrel{\widehat{=}} if\ binding' = \{\} \\
\qquad\qquad then\ SchemaRef\ ssname \\
\qquad\qquad else\ Application\ (\\
\qquad\qquad\qquad\qquad SchemaRef\ ssname, \\
\qquad\qquad\qquad\qquad BindingDisplay\ binding')) \\
\quad \rangle
\end{array}
$$

6.9.6 The Subsystem

In the following the treatment of mask variables is important.

Mask variables are in scope in subsystems of a masked subsystem. To achieve this effect we keep track of which variables are in scope, and we keep track of which of these variables are actually used. A subsystem is parameterised either if it is a masked subsystem with a non-empty set of mask variables, or if it makes any use of variables masked by subsystems which enclose the subsystem. The parameters to the subsystem definition will be the union of the variables it masks and those which are masked at higher levels and used in it.

The two fields of a subsystem, *subsys_info.mv_ctxt* and *subsys_info.used_maskvars* are used in this process. The former contains the set of variables masked at higher levels, and is given a value during the library look-up traversal which has already been completed. The latter contains the subset of those which are used in the subsystem or in the actual expressions supplied for the variables masked by this subsystem. This field is given its value during the subsystem translation traversal.

The *used_maskvars* value is obtained in the following way. First the union is formed of the used maskvars for each of the blocks in the subsystem. Then the variables masked by this subsystem are removed from this list, and the variables which occur in the mask variable context and are used in the actual parameters to the subsystem are added.

The additional schemas which arise from the presence of action subsystems (or in libraries, just in case the application involves action subsystems) are always parameterized in exactly the same way as if there were no action subsystems. This is not optimal, but saves the complication of keeping track of several more different sets of mask variables. This applies also to the top level definition for an action subsystem (which is always a schema expression).

The following specification is used for generating all the schemas involved, specification and invocation. A flag is also returned which is relevant only for held and reset schemas and indicates in effect whether there is any state in the subsystem. If set the schema definition may be suppressed (and the invocation field should be left empty).

z

$$
\begin{array}{l}
\boldsymbol{translate\_schema}: (DTYPE \times A\_SUBSYS \times \\
\qquad (PATH \times (\mathbb{F}\ (IDENT \times Z\_EXPR)) \times (\mathbb{F}\ PVALUE) \times \\
\qquad\qquad (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ PVALUE))) \\
\qquad \rightarrow (Z\_SPEC \times Z\_DECL \times BOOL) \\
\hline
\forall\ dtype: DTYPE;\ as: A\_SUBSYS;\ path: PATH;\ maskbinding: \mathbb{F}\ (IDENT \times Z\_EXPR); \\
\quad new\_maskvars,\ pused\_maskvars,\ bused\_maskvars: \mathbb{F}\ PVALUE; \\
\quad z\_spec: Z\_SPEC;\ inv: Z\_DECL;\ empty: BOOL; \\
\quad pack: PATH \times (\mathbb{F}\ (IDENT \times Z\_EXPR)) \times (\mathbb{F}\ PVALUE) \times \\
\qquad (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ PVALUE) \\
\mid pack = (path,\ maskbinding,\ new\_maskvars,\ pused\_maskvars,\ bused\_maskvars) \\
\bullet\ (dtype,\ as,\ pack) \mapsto (z\_spec,\ inv,\ empty) \in translate\_schema \\
\Leftrightarrow (\exists\ ssi: SUBSYS\_INFO;\ pi: PORT\_INFO; \\
\qquad\quad blks: PVALUE \twoheadrightarrow\!\!\!\!\!\!\shortmid\ \ A\_BLOCK;\ srefs: Z\_DECL;\ eqns: Z\_PRED; \\
\qquad\quad word: WORD;\ localname: IDENT \\
\qquad \bullet\ as = (subsys\_info \mathrel{\widehat=} ssi,\ port\_info \mathrel{\widehat=} pi,\ blocks \mathrel{\widehat=} blks) \\
\qquad \land\ word = \\
\qquad\qquad if\ dtype = DTActive\ then\ w_a(path2globw\ path) \\
\qquad\qquad else\ if\ dtype = DTHeld\ then\ w_h(path2globw\ path) \\
\qquad\qquad else\ if\ dtype = DTReset\ then\ w_r(path2globw\ path) \\
\qquad\qquad else\ path2globw\ path \\
\qquad \land\ localname = path2loci\ path \\
\qquad \land\ (srefs,\ empty) = schemadecs\ dtype\ blks \\
\qquad \land\ eqns = if\ dtype \in \{DTActive,\ DTPlain\} \\
\qquad\qquad then\ lines\_equations\ path\ blks\ ssi.lines\ (ssi.action\_info.complexes) \\
\qquad\qquad else\ PredConj\ \{\} \\
\qquad \land\ z\_spec = make\_ss\_spec\ word\ srefs\ eqns\ (new\_maskvars \cup bused\_maskvars) \\
\qquad \land\ inv = make\_ss\_inv\ localname\ word\ bused\_maskvars\ maskbinding)
\end{array}
$$

Note that in the following specification

z

$$
\begin{array}{|l}
\boldsymbol{translate\_action\_subsys}: \\
\quad (Z\_DECL \times Z\_DECL \\
\quad \times (PATH \times (\mathbb{F}\ (IDENT \times Z\_EXPR)) \times (\mathbb{F}\ PVALUE) \times \\
\qquad\qquad (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ PVALUE))) \\
\quad \rightarrow (Z\_SPEC \times Z\_DECL) \\
\hline
\forall\ adecl,\ sdecl: Z\_DECL;\ path: PATH;\ maskbinding: \mathbb{F}\ (IDENT \times Z\_EXPR); \\
\qquad new\_maskvars,\ pused\_maskvars,\ bused\_maskvars: \mathbb{F}\ PVALUE; \\
\qquad z\_spec: Z\_SPEC;\ inv: Z\_DECL \bullet \\
\ (adecl,\ sdecl,\ (path,\ maskbinding,\ new\_maskvars,\ pused\_maskvars,\ bused\_maskvars)) \\
\quad \mapsto (z\_spec,\ inv) \in translate\_action\_subsys \\
\Leftrightarrow \\
\quad (\exists\ word: WORD;\ localname: IDENT \\
\quad \bullet\ word = path2globw\ path \\
\quad \land\ localname = path2loci\ path \\
\quad \land\ z\_spec = make\_ass\_spec\ (word,\ adecl,\ sdecl,\ (new\_maskvars \cup bused\_maskvars)) \\
\quad \land\ inv = make\_ss\_inv\ localname\ word\ bused\_maskvars\ maskbinding)
\end{array}
$$

z

```
    $\boldsymbol{subsys\_spec\_and\_inv}: A\_SUBSYS \times (PATH \times (\mathbb{F}\ (IDENT \times Z\_EXPR))$
        $\times (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ PVALUE))$
    $\rightarrow Z\_SPEC \times INVOCATION$
────────────────
$\forall\ path: PATH; maskbinding: \mathbb{F}\ (IDENT \times Z\_EXPR);$
  $new\_maskvars, pused\_maskvars, bused\_maskvars: \mathbb{F}\ PVALUE;$
  $as: A\_SUBSYS; ssi, ssi': SUBSYS\_INFO; pi: PORT\_INFO;$
  $blks: PVALUE \twoheadrightarrow A\_BLOCK; srefs: Z\_DECL; eqns: Z\_PRED; word: WORD;$
  $maskbinding: \mathbb{F}\ (IDENT \times Z\_EXPR); hc: HOLD\_CONTEXT; localname: IDENT;$
  $new\_maskvars, pused\_maskvars, bused\_maskvars: \mathbb{F}\ PVALUE;$
  $z\_spec_h, z\_spec_r, z\_spec_a, z\_spec_s, z\_spec: Z\_SPEC;$
  $inv_h, inv_r, inv_a, inv_s, asinv: Z\_DECL; inv: INVOCATION;$
  $empty_h, empty_r, empty_a, empty: BOOL;$
  $pack: PATH \times (\mathbb{F}\ (IDENT \times Z\_EXPR)) \times (\mathbb{F}\ PVALUE)$
      $\times (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ PVALUE)$
|     $pack = (path, maskbinding, new\_maskvars, pused\_maskvars, bused\_maskvars)$
      $\wedge\ hc = as.subsys\_info.action\_info.held\_context$
      $\wedge\ localname = path2loci\ path$
      $\wedge\ (z\_spec_h, inv_h, empty_h) =$
              $if\ hc \in \{HCHeld, HCUnknown\}$
              $then\ translate\_schema\ (DTHeld, as, pack)$
              $else\ (\langle\rangle, \langle\rangle, true)$
      $\wedge\ (z\_spec_r, inv_r, empty_r) =$
              $if\ hc \in \{HCReset, HCUnknown\}$
              $then\ translate\_schema\ (DTReset, as, pack)$
              $else\ (\langle\rangle, \langle\rangle, true)$
      $\wedge\ (z\_spec_a, inv_a, empty_a) =\ translate\_schema\ ($
              $if\ ssi.action\_info.action\_subsys\ then\ DTActive\ else\ DTPlain, as, pack)$
      $\wedge\ asinv =$    $if$      $hc = HCHeld$
                     $then$    $if\ empty_h\ then\ \langle\rangle\ else\ inv_h$
                     $else$    $if\ empty_r\ then\ \langle\rangle\ else\ inv_r$
      $\wedge\ (z\_spec_s, inv_s) =$
              $if\ ssi.action\_info.action\_subsys$
              $then\ translate\_action\_subsys\ (inv_a, asinv,\ \ pack)$
              $else\ (\langle\rangle, inv_a)$
      $\wedge\ z\_spec = \frown/\ \langle if\ empty_h\ then\ \langle\rangle\ else\ z\_spec_h,$
              $if\ empty_r\ then\ \langle\rangle\ else\ z\_spec_r, z\_spec_a, z\_spec_s\rangle$
      $\wedge\ inv = (OtherInv\ localname, (inv_s, if\ empty_h\ then\ \langle\rangle\ else\ inv_h,$
              $if\ empty_r\ then\ \langle\rangle\ else\ inv_r))$
$\bullet\ subsys\_spec\_and\_inv\ (as, pack) = (z\_spec, inv)$
```

In *translate_subsystem* variables concerned with masked variables are used as follows:

maskbinding this is the information necessary to construct a binding of actual parameters to the subsystem (as a map from identifiers to Z expressions)

new_maskvars if this subsystem is masked then *new_maskvars* is the set of the names of variables masked by it, otherwise it is empty

pused_maskvars this is the set of names of variables which are masked in higher level subsystems and used in the actual parameters to this subsystem

bused_maskvars this is the set of names of variables which are masked in higher level subsystems and used in the actual parameters to blocks within this subsystem (excepting usages which are hidden by a nested mask)

z

$$
\begin{array}{l}
\textbf{translate\_subsystem}: PATH \rightarrow A\_BLOCK \rightarrowtail\!\!\!\!\rightarrow A\_BLOCK \\
\hline
\forall\ path: PATH;\ ab,\ ab': A\_BLOCK \bullet \\
(ab \mapsto ab') \in translate\_subsystem\ path \\
\Leftrightarrow (\exists\ as: A\_SUBSYS;\ ssi,\ ssi': SUBSYS\_INFO;\ pi: PORT\_INFO; \\
\quad blks: PVALUE \twoheadrightarrow\!\!\!\!\!\!\!\!\!\shortmid\;\; A\_BLOCK;\ srefs: Z\_DECL;\ eqns: Z\_PRED; \\
\quad word: WORD;\ localname: IDENT; \\
\quad maskbinding: \mathbb{F}\ (IDENT \times Z\_EXPR);\ hc: HOLD\_CONTEXT; \\
\quad new\_maskvars,\ pused\_maskvars,\ bused\_maskvars: \mathbb{F}\ PVALUE; \\
\quad z\_spec_h,\ z\_spec_r,\ z\_spec_a,\ z\_spec_s,\ z\_spec: Z\_SPEC; \\
\quad inv_h,\ inv_r,\ inv_a,\ inv_s: Z\_DECL;\ inv: INVOCATION; \\
\quad empty_h,\ empty_r,\ empty_a,\ empty: BOOL; \\
\quad pack: PATH \times (\mathbb{F}\ (IDENT \times Z\_EXPR)) \times (\mathbb{F}\ PVALUE) \times \\
\qquad (\mathbb{F}\ PVALUE) \times (\mathbb{F}\ PVALUE) \\
\mid\ pack = (path,\ maskbinding,\ new\_maskvars,\ pused\_maskvars,\ bused\_maskvars) \\
\bullet\ ab = ASubsys\ as \\
\wedge\ as = (subsys\_info \mathrel{\widehat{=}} ssi,\ port\_info \mathrel{\widehat{=}} pi,\ blocks \mathrel{\widehat{=}} blks) \\
\wedge\ (maskbinding,\ new\_maskvars,\ pused\_maskvars) \\
\qquad = make\_maskbinding\ (ssi.subpars,\ ssi.mv\_ctxt) \\
\wedge\ bused\_maskvars = (\bigcup\ \{blk: \operatorname{ran}\ blks \bullet get\_used\_maskvars\ blk\}) \setminus new\_maskvars \\
\wedge\ (z\_spec,\ inv) = subsys\_spec\_and\_inv\ (as,\ pack) \\
\wedge\ ssi' = \quad (subpars \mathrel{\widehat{=}} ssi.subpars,\ syspars \mathrel{\widehat{=}} ssi.syspars,\ lines \mathrel{\widehat{=}} ssi.lines, \\
\qquad specification \mathrel{\widehat{=}} z\_spec,\ invocation \mathrel{\widehat{=}} inv,\ virtual \mathrel{\widehat{=}} VInhibit, \\
\qquad used\_maskvars \mathrel{\widehat{=}} bused\_maskvars \cup pused\_maskvars, \\
\qquad mv\_ctxt \mathrel{\widehat{=}} ssi.mv\_ctxt,\ action\_info \mathrel{\widehat{=}} ssi.action\_info) \\
\wedge\ ab' = ASubsys\ (subsys\_info \mathrel{\widehat{=}} ssi',\ port\_info \mathrel{\widehat{=}} pi,\ blocks \mathrel{\widehat{=}} blks))
\end{array}
$$

z

$$\boldsymbol{translate\_system}: A\_BLOCK \nrightarrow A\_BLOCK$$

$$\begin{array}{l}
\forall\ ab,\ ab': A\_BLOCK \bullet \\
ab \mapsto ab' \in translate\_system \\
\Leftrightarrow \\
(\exists\ pv: PVALUE;\ as: A\_SUBSYS \\
\bullet\ \ ab = ASubsys\ as \\
\land\ (name \mathrel{\widehat{=}} Name,\ value \mathrel{\widehat{=}} pv) \in as.subsys\_info.syspars \\
\land\ ab \mapsto ab' \in a\_block\_map\ translate\_subsystem\ \langle pv \rangle)
\end{array}$$

6.10 Metadata Extraction

When translating a library it is necessary to produce a metadata file showing how the masked subsystems in the library can be invoked. The meta-element for a masked subsystem will use the *SourceBlock* parameter value to uniquely identify the library block referred to (which will therefore be the sole *SelectionParameter*), and will provide for each block:

1. the Z specification name which must be used to invoke the block
2. a set of *TransmittedParameters* which includes each of the *MaskVariables* with translation code SVM.

Whether the translation code SVM is the best one to use is moot, and the optimum may vary according to the usage of the masked subsystem and its content. It is open to the user to edit the metadata and substitute a translation code which works better for his application.

z

$$[C, R]$$

$$\boldsymbol{libmap\_null}: R \rightarrow C \rightarrow A\_LIB\_BLOCK \rightarrow R \times A\_LIB\_BLOCK$$

$$\begin{array}{l}
\forall\ r: R;\ c: C;\ alb: A\_LIB\_BLOCK \bullet \\
libmap\_null\ r\ c\ alb = (r,\ alb)
\end{array}$$

path2pv converts a *PATH* (which is a sequence of *PVALUE*s) to a single *PVALUE*, using "/" as separator.

z

$$\begin{array}{|l}
\quad \boldsymbol{path2pv}: PATH \rightarrow PVALUE \\
\hline
\quad \forall\ pv1, pv2: PVALUE;\ w: seq\ PVALUE \bullet \\
\quad path2pv\ \langle\rangle = sc2pv\ \texttt{""} \\
\land \quad path2pv\ \langle pv1 \rangle = pv1 \\
\land \quad path2pv\ (\langle pv1 \rangle \frown \langle pv2 \rangle \frown w) \\
\quad = sc2pv\ (\frown/\ \langle pv2sc\ pv1, \texttt{"/"}, pv2sc\ (path2pv\ (\langle pv2 \rangle \frown w)) \rangle)
\end{array}$$

z

$$\begin{array}{|l}
\quad \boldsymbol{get\_maskstyles}: \mathbb{F}\ PARAM \rightarrow\!\!\!\!\mapsto (PVALUE \twoheadrightarrow\!\!\!\!\mapsto MASK\_STYLE) \\
\hline
\forall\ pars: \mathbb{F}\ PARAM \bullet \\
\quad dom\ get\_maskstyles = \\
\qquad \{pars: \mathbb{F}\ PARAM \\
\qquad \mid sc2pn\ (\!| \{\texttt{"MaskVariables"}, \texttt{"MaskStyleString"}\} |\!) \\
\qquad\quad \subseteq dom\ (param\_value\ pars)\} \\
\land \quad get\_maskstyles\ pars = \\
\quad (\mu\ mvars: seq\ PVALUE;\ mstyles: seq\ MASK\_STYLE \\
\quad \mid mvars = \\
\qquad if\ sc2pn\ \texttt{"MaskVariables"} \in dom\ (param\_value\ pars) \\
\qquad then\ parse\_maskvar\_param\ (param\_value\ pars\ (sc2pn\ \texttt{"MaskVariables"})) \\
\qquad else\ \{\} \\
\quad \land\ mstyles = \\
\qquad if\ sc2pn\ \texttt{"MaskStyleString"} \in dom\ (param\_value\ pars) \\
\qquad then\ parse\_maskstyle\_param\ (param\_value\ pars\ (sc2pn\ \texttt{"MaskStyleString"})) \\
\qquad else\ \{\} \\
\quad \bullet\ \{n: dom\ mvars \bullet mvars\ n \mapsto mstyles\ n\})
\end{array}$$

z

$$\begin{array}{|l}
\quad \boldsymbol{maskstyle2pv}: MASK\_STYLE \rightarrow PVALUE \\
\hline
\quad maskstyle2pv\ MSEdit = sc2pv\ \texttt{"SVM"} \\
\land \quad maskstyle2pv\ MSCheckbox = sc2pv\ \texttt{"Checkbox"} \\
\land\ (\forall spv: seq\ PVALUE \bullet \\
\quad maskstyle2pv\ (MSPopup\ spv) = (parse\_popup\_tc^{\sim})\ spv)
\end{array}$$

The following converts a *PORT_DETAILS* map to a sequence of *PORT_TYPE*s. If the domain is not an initial segment of ℕ then the empty sequence is returned.

z

$$\begin{array}{|l}
\textbf{details2types}: (PVALUE \nrightarrow PORT\_DETAILS) \rightarrow (seq\ PORT\_TYPE) \\
\hline
\forall\ pds: PVALUE \nrightarrow PORT\_DETAILS;\ pts : \mathbb{N} \nrightarrow PORT\_TYPE \\
\mid pts = num2pvalue \fatsemi pds \fatsemi (\lambda pd{:}PORT\_DETAILS \bullet pd.port\_type) \\
\bullet\ details2types\ pds = if\ \ pts \in (seq\ PORT\_TYPE)\ then\ pts\ else\ \langle\rangle
\end{array}$$

In the following specification *rootp* is the path to the top level subsystem of an artificial subsystem, and *relp* is the path to the subsystem under consideration from that point. If an artificial subsystem is not being processed then *as_name* will be *NullString* and *rootp* will be a singleton list containing the system name.

z

$$\begin{array}{|l}
\textbf{masksubsys\_metaelem}: PVALUE \times PATH \rightarrow PATH \rightarrow A\_SUBSYS \\
\qquad \rightarrow META\_ELEMENT \\
\hline
\forall\ as\_name: PVALUE;\ rootp,\ relp: PATH;\ ass: A\_SUBSYS; \\
\quad pars,\ mtps: \mathbb{F}\ PARAM;\ sb: PVALUE;\ mvs: \mathbb{F}\ PVALUE; \\
\quad META\_ELEMENT;\ maskstyles: PVALUE \nrightarrow MASK\_STYLE \\
\mid pars = ass.subsys\_info.subpars \\
\land\ sb = path2pv\ (rootp \frown relp) \\
\land\ maskstyles = get\_maskstyles\ pars \\
\land\ mtps = \{mv: dom\ maskstyles \bullet (\\
\qquad name \mathrel{\widehat=} sc2pn\ (pv2sc\ mv), \\
\qquad value \mathrel{\widehat=} maskstyle2pv\ (maskstyles\ mv))\} \\
\land\ z\_name = path2globi \\
\quad ((if\ \ as\_name = NullString\ then\ rootp\ else\ \langle as\_name\rangle) \frown relp) \\
\land\ held\_z\_name = \quad if\ \ (second \fatsemi held\_inv)\ ass.subsys\_info.invocation = \langle\rangle \\
\qquad then\ \ Nil \\
\qquad else\ \ Value\ (i_h\ \ z\_name) \\
\land\ reset\_z\_name = \quad if\ \ (second \fatsemi reset\_inv)\ ass.subsys\_info.invocation = \langle\rangle \\
\qquad then\ \ Nil \\
\qquad else\ \ Value\ (i_r\ \ z\_name) \\
\land\ block\_path = \langle StarPat\rangle \\
\land\ select\_pars = \{(name \mathrel{\widehat=} sc2pn\ \texttt{"}SourceBlock\texttt{"},\ value \mathrel{\widehat=} sb)\} \\
\land\ transmit\_pars = mtps \\
\land\ input\_port\_types = details2types\ \ ass.port\_info.input\_port\_details \\
\land\ output\_port\_types = details2types\ \ ass.port\_info.output\_port\_details \\
\land\ used\_maskvars = \quad (\bigcup \{blk: ran\ (ass.blocks) \\
\qquad \bullet\ get\_used\_maskvars\ blk\}) \setminus (dom\ maskstyles) \\
\bullet\ masksubsys\_metaelem\ (as\_name,\ rootp)\ relp\ ass = \theta META\_ELEMENT
\end{array}$$

z

$$
\begin{array}{l}
\boldsymbol{ssmap\_metadata}: PVALUE \times PATH \rightarrow PATH \\
\qquad \rightarrow A\_SUBSYS \times (PVALUE \nrightarrow \mathbb{F}\ META\_ELEMENT) \\
\qquad \rightarrow (\mathbb{F}\ META\_ELEMENT) \times A\_SUBSYS \\
\hline
\forall\ as\_name: PVALUE;\ rootp,\ relp: PATH;\ ass: A\_SUBSYS; \\
\quad mdmap: PVALUE \nrightarrow \mathbb{F}\ META\_ELEMENT \bullet \\
ssmap\_metadata\ (as\_name,\ rootp)\ relp\ (ass,\ mdmap) = \\
\qquad (\{masksubsys\_metaelem\ (as\_name,\ rootp)\ relp\ ass\} \cup \bigcup(ran\ mdmap),\ ass)
\end{array}
$$

z

$$
\begin{array}{l}
\boldsymbol{extract\_metadata}: PVALUE \times PATH \rightarrow A\_BLOCK \nrightarrow META\_FILE \\
\hline
\forall\ as\_name: PVALUE;\ rootp: PATH;\ ab: A\_BLOCK;\ ass: A\_SUBSYS; \\
\quad mes: \mathbb{F}\ META\_ELEMENT \\
\mid ASubsys\ ass = ab \\
\land\ (mes,\ ab) = a\_block\_map\_cr \\
\qquad (libmap\_null\ \{\},\ ssmap\_metadata\ (as\_name,\ rootp),\ newc\_path) \\
\qquad \langle\rangle\ ab \\
\bullet\ ran(extract\_metadata\ (as\_name,\ rootp)\ ab) = mes
\end{array}
$$

6.11 Generating Artificial Subsystems

The *A_BLOCK* produced from *SYSTEM* is now transformed to create various artificial subsystems which are also *A_BLOCK*s.

Each artificial subsystem specification determines a subsystem of the main model which will yield a "top level" subsystem for the artificial subsystem, and a set of modifications to subsystems or library block references which are contained (at any depth) within that subsystem.

There are two kinds of modification which are permitted in this process. Any subsystem, including the top level subsystem may be subjected to a filter, which is a prescription, in effect, that some of the blocks in the subsystem are to be discarded. Any block reference may be directed at some artificial subsystem of the library in which the original reference was satisfied. There two kinds of modification will be referred to as filters and qualifications respectively. The type *BLOCK_MODIFIER* (artificial subsystem modification) supplies information about a single modification (which may either be a filter or a qualification). When combined with a relative path which identifies a block within the artificial subsystem an *BLOCK_MODIFIER* becomes an *BLOCK_MODIFIER_SPEC* and the complete description of an artificial subsystem consists of the identification of the top level subsystem and a filter on that top level subsystem, together with a sequence of *BLOCK_MODIFIER_SPEC*s.

The following function derives from a sequence of *BLOCK_MODIFIER_SPEC*s the relevant modification information for some path. The information returned will be the *BLOCK_MODIFIER* associated with the path, if there is one.

If a block is modified or contains a modified block then it must be transcribed for re-translation (in the latter case only the full names of some subsystems will have changed, local names will be unchanged) otherwise it can be referred to without change of name or substance.

The implementation may detect and report as "Error"s the presence of multiple modifications applicable to a single block.

z

$$\mathbf{get\_as\_mod}: seq\ BLOCK\_MODIFIER\_SPEC \to PATH \\ \nrightarrow BLOCK\_MODIFIER$$

$$\forall\ sss: seq\ BLOCK\_MODIFIER\_SPEC; p{:}PATH; f{:}BLOCK\_MODIFIER \bullet \\ (p \mapsto f) \in get\_as\_mod\ sss \\ \Leftrightarrow \\ (\exists pat: PATTERN \\ \bullet \quad (path \mathrel{\widehat{=}} pat, filter \mathrel{\widehat{=}} f) \in ran\ sss \\ \quad \land\ p \mapsto pat \in path\_match)$$

The function is packaged separately for use specifically with respect to subsystems and block references. The permitted modifications for subsystems and block references differ, and if the modification requested (in the *BLOCK_MODIFIER_SPEC*s) is inappropriate for the kind of block under consideration this should be reported as an "Error". i.e. raise "Error" if *get_as_filter* finds a qualification or if *get_as_qualification* finds a filter.

z

$$\mathbf{get\_as\_filter}: seq\ BLOCK\_MODIFIER\_SPEC \to PATH \\ \nrightarrow BLOCK\_MODIFIER$$

$$\forall\ sss: seq\ BLOCK\_MODIFIER\_SPEC; p{:}PATH; f{:}BLOCK\_MODIFIER \bullet \\ (p \mapsto f) \in get\_as\_filter\ sss \\ \Leftrightarrow \\ (p \mapsto f) \in get\_as\_mod\ sss \\ \land\ (\exists\ spv: \mathbb{F}\ PVALUE \bullet\ f = Include\ spv \lor f = Exclude\ spv)$$

z

$$\mathbf{get\_as\_qualification}: seq\ BLOCK\_MODIFIER\_SPEC \to PATH \\ \nrightarrow PVALUE$$

$$\forall\ sss: seq\ BLOCK\_MODIFIER\_SPEC; p{:}PATH; as\_name: PVALUE \bullet \\ (p \mapsto as\_name) \in get\_as\_qualification\ sss \\ \Leftrightarrow \\ (p \mapsto ASname\ as\_name) \in get\_as\_mod\ sss$$

The translation of an artificial subsystem involves copying part of the original system making the following changes:

- if a block is to be referred to rather than retranslated it is replaced by a stub, the purpose of which is to supply the declaration for invocation of the block. This applies also to library blocks (except port blocks, see below).
- otherwise the block is transcribed, implementing any specified filter and removing the invocation and specification fields (which will be recomputed).
- port blocks which are not filtered out are left unchanged (i.e. they are not converted into a stub).

The following specifies how a subsystem stub is created.

z

```
│     create_ass_stub: A_SUBSYS → A_BLOCK
├──────────────────────────
│ ∀ ass: A_SUBSYS•
│     create_ass_stub ass = ASubsys (
│         subsys_info ≘ (subpars ≘ {}, syspars ≘ {},
│             lines ≘ {}, specification ≘ ⟨⟩,
│             invocation ≘ ass.subsys_info.invocation,
│             virtual ≘ ass.subsys_info.virtual,
│             used_maskvars ≘ ass.subsys_info.used_maskvars, mv_ctxt ≘ {},
│             action_info ≘ initial_action_info HCUnknown),
│         port_info ≘ ass.port_info,
│         blocks ≘ {})
```

The following specifies how a library block stub is created.

z

```
│     create_lib_stub: A_LIB_BLOCK → A_BLOCK
├──────────────────────────
│ ∀ alb: A_LIB_BLOCK•
│     create_lib_stub alb = ALibBlock (
│         block_info ≘ (pars ≘ alb.block_info.pars,
│             specification ≘ ⟨⟩,
│             invocation ≘ alb.block_info.invocation,
│             virtual ≘ alb.block_info.virtual,
│             used_maskvars ≘ alb.block_info.used_maskvars,
│             input_port_types ≘ {},
│             output_port_types ≘ {}),
│         port_info ≘ alb.port_info)
```

The following specifies how the blocks in a subsystem are filtered. It is a partially specified total function which is not used outside the part of the domain which is specified (i.e. not used for block reference qualifications). A similar comment applies to some of the functions which directly or indirectly call this function.

z

$$
\begin{array}{|l}
\quad \boldsymbol{filter\_blocks}: BLOCK\_MODIFIER \to (PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} A\_BLOCK) \\
\quad \to (PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} A\_BLOCK) \\
\hline
\forall\, f: BLOCK\_MODIFIER;\ blocks,\ blocks': (PVALUE \mathrel{+\!\!\!\!\twoheadrightarrow} A\_BLOCK);\ pvs: \mathbb{F}\ PVALUE \\
\quad |\quad f = Include\ pvs \land blocks' = pvs \lhd blocks \\
\quad\ \lor\quad f = Exclude\ pvs \land blocks' = pvs \mathbin{⩤} blocks \\
\quad \bullet\ filter\_blocks\ f\ blocks = blocks'
\end{array}
$$

The following function is used to discard the blocks which are to be filtered out of a subsystem.

z

$$
\begin{array}{|l}
\quad \boldsymbol{filter\_subsys}: BLOCK\_MODIFIER \to A\_SUBSYS \to A\_SUBSYS \\
\hline
\forall\, f: BLOCK\_MODIFIER;\ ass: A\_SUBSYS \bullet \\
\quad filter\_subsys\ f\ ass = \\
\qquad (\quad subsys\_info \mathrel{\widehat{=}} ass.subsys\_info, \\
\qquad\quad port\_info \mathrel{\widehat{=}} ass.port\_info, \\
\qquad\quad blocks \mathrel{\widehat{=}} filter\_blocks\ f\ ass.blocks)
\end{array}
$$

The following functions transcribe and translate the subsystems which are changed in some artificial subsystem. Note that these functions do nothing sensible if supplied with a path which is not in the specified subsystem (the *path_change* function will not yield defined results in that case).

z

$$
\begin{array}{|lll}
\boldsymbol{CHANGE\_INFO} & ::= & \boldsymbol{Unchanged} \\
 & | & \boldsymbol{Changed}
\end{array}
$$

Given the results of filtering the blocks in a subsystem the following function specifies how the filtered subsystem is to be constructed. This involves reconstructing the action complexes, repeating the checks on the complexes which might raise new errors because of the filtering operation.

The *port_info* is recomputed discarding information about ports which have been deleted. The following two functions determine which ports remain:

z

$$\boldsymbol{inport\_names\_remaining}: (PVALUE \ffun A\_BLOCK) \rightarrow \mathbb{F}\ PVALUE$$

$$\begin{array}{l} \forall\ blocks: PVALUE \ffun A\_BLOCK \bullet \\ inport\_names\_remaining\ blocks = \\ \qquad \{pv : dom\ blocks;\ alb: A\_LIB\_BLOCK;\ pn: PVALUE \\ \qquad \mid ALibBlock\ alb = blocks\ pv \\ \qquad \land (name \mathrel{\widehat{=}} BlockType,\ value \mathrel{\widehat{=}} InPort) \in alb.block\_info.pars \\ \qquad \land (name \mathrel{\widehat{=}} Port,\ value \mathrel{\widehat{=}} pn) \in alb.block\_info.pars \\ \qquad \bullet\ pn\} \end{array}$$

z

$$\boldsymbol{outport\_names\_remaining}: (PVALUE \ffun A\_BLOCK) \rightarrow \mathbb{F}\ PVALUE$$

$$\begin{array}{l} \forall\ blocks: PVALUE \ffun A\_BLOCK \bullet \\ outport\_names\_remaining\ blocks = \\ \qquad \{pv : dom\ blocks;\ alb: A\_LIB\_BLOCK;\ pn: PVALUE \\ \qquad \mid ALibBlock\ alb = blocks\ pv \\ \qquad \land (name \mathrel{\widehat{=}} BlockType,\ value \mathrel{\widehat{=}} OutPort) \in alb.block\_info.pars \\ \qquad \land (name \mathrel{\widehat{=}} Port,\ value \mathrel{\widehat{=}} pn) \in alb.block\_info.pars \\ \qquad \bullet\ pn\} \end{array}$$

This specification prescribes how to construct a filtered subsystem given the results of modifying as necessary the blocks in the subsystem.

z

```
    make_filtered_subsys:
    (PATH × (PVALUE ⇸⇸ A_BLOCK × CHANGE_INFO)
        × A_SUBSYS × CHANGE_INFO)
    → (A_SUBSYS × CHANGE_INFO)
────────────────────────────────
∀ path:PATH; ass, ass′: A_SUBSYS; ci, ci′: CHANGE_INFO;
  ssi, ssi′: SUBSYS_INFO;
  pi, pi′: PORT_INFO; blocks, blocks′: PVALUE ⇸⇸ A_BLOCK;
  blockcs: PVALUE ⇸ (A_BLOCK × CHANGE_INFO)
| ass = (subsys_info ≙ ssi, port_info ≙ pi, blocks ≙ blocks)
∧ blocks′ = blockcs ⨾ first
∧ ci′ =        if ran (blockcs ⨾ second) = {Unchanged}
               then ci
               else Changed
∧ ssi′ =
        if ci′ = Changed
        then (subpars ≙ ssi.subpars,
               syspars ≙ ssi.syspars,
               lines ≙ ssi.lines,
               specification ≙ ⟨⟩,
               invocation ≙ (NoInv, (⟨⟩, ⟨⟩, ⟨⟩)),
               virtual ≙ VUnknown,
               used_maskvars ≙ ssi.used_maskvars,
               mv_ctxt ≙ ssi.mv_ctxt,
               action_info ≙ (
                    action_subsys ≙ ssi.action_info.action_subsys,
                    complexes ≙ make_action_complexes (ssi.lines, blocks′, path),
                    held_context ≙ ssi.action_info.held_context))
        else ssi
∧ pi′ = (input_port_details ≙ (inport_names_remaining blocks′) ◁ pi.input_port_details,
        output_port_details ≙ (outport_names_remaining blocks′) ◁ pi.output_port_details)
∧ ass′ = (subsys_info ≙ ssi′, port_info ≙ pi′, blocks ≙ blocks′)
• make_filtered_subsys (path, blockcs, ass, ci) = (ass′, ci′)
```

The following specifications cover the modification of library blocks. In all cases except block references a library stub is created which results in reference to the library invocation produced for the main model.

The first part of the specificatiom covers how a qualified library block is processed. This involves first checking that the qualified block is a block reference. If the block is indeed a block reference then a library lookup is undertaken, qualified by the artificial subsystem name given in the qualification, and a new library block obtained in this way is returned. If a library lookup is attempted and fails an *Error* should be raised.

z

$$
\begin{array}{l}
\textbf{qualify\_blockref}: META\_FILE \rightarrow ART\_SUBSYS\_SPEC \rightarrow PATH \rightarrow PATH \\
\quad \rightarrow HOLD\_CONTEXT \rightarrow (\mathbb{F}\ PVALUE) \rightarrow A\_LIB\_BLOCK \nrightarrow A\_LIB\_BLOCK \\
\hline
\forall\ mf: META\_FILE;\ art: ART\_SUBSYS\_SPEC;\ toppath, relpath: PATH; \\
\quad alb, alb': A\_LIB\_BLOCK;\ hc: HOLD\_CONTEXT; \\
\quad bi, bi': BLOCK\_INFO;\ pi: PORT\_INFO; \\
\quad ci: CHANGE\_INFO;\ maskvars: \mathbb{F}\ PVALUE;\ hc: HOLD\_CONTEXT \\
\mid alb = (block\_info \mathrel{\widehat{=}} bi, port\_info \mathrel{\widehat{=}} pi) \\
\land alb' = (block\_info \mathrel{\widehat{=}} bi', port\_info \mathrel{\widehat{=}} pi) \\
\bullet\ alb \mapsto alb' \in qualify\_blockref\ mf\ art\ toppath\ relpath\ hc\ maskvars \\
\Leftrightarrow (name \mathrel{\widehat{=}} BlockType, value \mathrel{\widehat{=}} Reference) \in bi.pars \\
\quad \land (\exists\ as\_name: PVALUE \bullet \\
\qquad relpath \mapsto as\_name \in get\_as\_qualification\ art.rest \\
\qquad \land instantiate\_last\_match\ as\_name\ mf\ hc\ (\langle art.name \rangle \frown relpath)\ maskvars\ bi.pars \\
\qquad\quad = IMatch\ bi' \\
\quad)
\end{array}
$$

If qualification fails, a library stub is used.

z

$$
\begin{array}{l}
\textbf{make\_modified\_libblock}: META\_FILE \rightarrow ART\_SUBSYS\_SPEC \rightarrow PATH \\
\quad \rightarrow PATH \rightarrow HOLD\_CONTEXT \rightarrow (\mathbb{F}\ PVALUE) \rightarrow A\_LIB\_BLOCK \\
\quad \rightarrow (A\_LIB\_BLOCK \times CHANGE\_INFO) \\
\hline
\forall\ mf: META\_FILE;\ art: ART\_SUBSYS\_SPEC;\ toppath, relpath: PATH; \\
\quad ci: CHANGE\_INFO;\ hc: HOLD\_CONTEXT;\ maskvars: \mathbb{F}\ PVALUE; \\
\quad alb, alb': A\_LIB\_BLOCK \\
\mid alb \in dom\ (qualify\_blockref\ mf\ art\ toppath\ relpath\ hc\ maskvars) \\
\qquad\qquad \land alb' = qualify\_blockref\ mf\ art\ toppath\ relpath\ hc\ maskvars\ alb \\
\qquad\qquad \land ci = Changed \\
\quad \lor alb \notin dom\ (qualify\_blockref\ mf\ art\ toppath\ relpath\ hc\ maskvars) \\
\qquad\qquad \land ALibBlock\ alb' = create\_lib\_stub\ alb \\
\qquad\qquad \land ci = Unchanged \\
\bullet\ make\_modified\_libblock\ mf\ art\ toppath\ relpath\ hc\ maskvars\ alb = (alb', ci)
\end{array}
$$

We now specify the filtering operation by mutual recursion.

z

$$
\begin{array}{l}
\boldsymbol{artificial\_subsys\_filter}: META\_FILE \rightarrow ART\_SUBSYS\_SPEC \rightarrow PATH \\
\quad \rightarrow PATH \rightarrow A\_SUBSYS \rightarrow (A\_SUBSYS \times CHANGE\_INFO); \\
\boldsymbol{artificial\_subsys\_mapfilter}: META\_FILE \\
\quad \rightarrow (ART\_SUBSYS\_SPEC \times PATH \times PATH \times A\_SUBSYS \times CHANGE\_INFO) \\
\quad \rightarrow (A\_SUBSYS \times CHANGE\_INFO) \\
\hline
(\forall mf: META\_FILE; art: ART\_SUBSYS\_SPEC; ts: BLOCK\_MODIFIER\_SPEC; \\
\quad sss: seq\ BLOCK\_MODIFIER\_SPEC; toppath, relpath, newpath: PATH; \\
\quad ass, ass', ass'', ass''': A\_SUBSYS; f{:}BLOCK\_MODIFIER; \\
\quad ci, ci': CHANGE\_INFO \\
\mid ts = (path \mathrel{\widehat{=}} \langle\rangle, filter \mathrel{\widehat{=}} art.top.filter) \land sss = \langle ts \rangle \frown art.rest \\
\land (relpath \in dom\ (get\_as\_filter\ sss) \\
\quad \land ass' = filter\_subsys\ (get\_as\_filter\ sss\ relpath)\ ass \\
\quad \land ci = Changed \\
\quad \lor relpath \notin dom\ (get\_as\_filter\ sss) \land ass' = ass) \land ci = Unchanged \\
\land (ass'', ci') = artificial\_subsys\_mapfilter\ mf\ (art, toppath, relpath, ass', ci) \\
\land newpath = \langle art.name \rangle \frown relpath \\
\land ASubsys\ ass''' = \ if\ ci' = Changed \\
\qquad\qquad then\ translate\_subsystem\ newpath\ (ASubsys\ ass'') \\
\qquad\qquad else\ create\_ass\_stub\ ass \\
\bullet\ artificial\_subsys\_filter\ mf\ art\ toppath\ relpath\ ass = (ass''', ci')) \\
\land \quad (\forall mf: META\_FILE; art: ART\_SUBSYS\_SPEC; toppath, relpath{:}PATH; \\
\qquad ci, ci': CHANGE\_INFO; cis: \mathbb{F}\ CHANGE\_INFO; ssi: SUBSYS\_INFO; \\
\qquad pi: PORT\_INFO; blocks, blocks': PVALUE \twoheadrightarrow\!\!\!\!\!\shortmid\;\ A\_BLOCK; \\
\qquad blockcs: PVALUE \nrightarrow (A\_BLOCK \times CHANGE\_INFO); \\
\qquad hc: HOLD\_CONTEXT; maskvars: \mathbb{F}\ PVALUE; ass, ass': A\_SUBSYS \\
\quad \mid ass = (subsys\_info \mathrel{\widehat{=}} ssi, port\_info \mathrel{\widehat{=}} pi, blocks \mathrel{\widehat{=}} blocks) \\
\quad \land hc = ssi.action\_info.held\_context \land maskvars = ssi.mv\_ctxt \\
\quad \land blockcs = \{pv: dom\ blocks; ab, ab'{:}A\_BLOCK; alb, alb'{:}A\_LIB\_BLOCK; \\
\qquad ass, ass'{:}A\_SUBSYS; lci{:}CHANGE\_INFO \\
\qquad \mid\ ab = blocks\ pv \\
\qquad \land\ (ab = ALibBlock\ alb \land ab' = ALibBlock\ alb' \\
\qquad\quad \land (alb', lci) \\
\qquad\qquad = make\_modified\_libblock\ mf\ art\ toppath\ (relpath \frown \langle pv \rangle)\ hc\ maskvars\ alb \\
\qquad \lor ab = ASubsys\ ass \land ab' = ASubsys\ ass' \\
\qquad\quad \land (ass', lci) = artificial\_subsys\_filter\ mf\ art\ toppath\ (relpath \frown \langle pv \rangle)\ ass) \\
\qquad \bullet\ pv \mapsto (ab', lci)\} \\
\quad \land (ass', ci') = make\_filtered\_subsys\ (\langle art.name \rangle \frown relpath, blockcs, ass, ci) \\
\quad \bullet\ artificial\_subsys\_mapfilter\ mf\ (art, toppath, relpath, ass, ci) = (ass', ci'))
\end{array}
$$

z

$$
\begin{array}{l}
\boldsymbol{make\_artificial\_subsys\_block}: META\_FILE \rightarrow BOOL \rightarrow ART\_SUBSYS\_SPEC \\
\qquad \rightarrow A\_BLOCK \nrightarrow A\_BLOCK \times META\_FILE \\
\hline
\forall\ lib: BOOL;\ art: ART\_SUBSYS\_SPEC;\ ab,\ ab': A\_BLOCK;\ imf,\ mf: META\_FILE \bullet \\
\quad ab \mapsto (ab',\ mf) \in make\_artificial\_subsys\_block\ imf\ lib\ art \\
\quad \Leftrightarrow \\
\quad (\exists_1\ ass: A\_SUBSYS;\ path: PATH \bullet \\
\quad\ \ (path,\ art.top.path) \in path\_match \\
\quad \land\ ASubsys\ ass = a\_block\_select\ (tail\ path)\ ab \\
\quad \land\ ab' = ASubsys\ ((artificial\_subsys\_filter\ imf\ art\ path\ \langle\rangle\ ass).1) \\
\quad \land\ mf = if\ lib\ then\ extract\_metadata\ (art.name,\ path)\ ab'\ else\ \langle\rangle)
\end{array}
$$

7 TRANSCRIBING SPECIFICATIONS TO OUTPUT FILES

7.1 Extracting The Specifications

The transcription of output from an *A_BLOCK* to a *SYS_SPEC* is defined.

z

$$
\begin{array}{l}
\quad \boldsymbol{a\_block\_to\_sys\_spec}: A\_BLOCK \rightarrow SYS\_SPEC \\
\hline
\quad \forall\ ab: A\_BLOCK \bullet \\
\quad (\exists\ alb: A\_LIB\_BLOCK \bullet\ ab = ALibBlock\ alb \\
\quad \Rightarrow a\_block\_to\_sys\_spec\ ab = SysSpec\ (\\
\qquad\qquad block\_specs \mathrel{\widehat{=}} \{\}, \\
\qquad\qquad z\_spec \mathrel{\widehat{=}} alb.block\_info.specification)) \\
\land \quad (\exists\ ass: A\_SUBSYS;\ bs: BLOCK\_SPECS \bullet\ ab = ASubsys\ ass \\
\quad \land\ bs = \{pv{:}PVALUE;\ ss: SYS\_SPEC;\ ab2: A\_BLOCK \\
\qquad\quad |\ (pv \mapsto ab2) \in ass.blocks \\
\qquad\quad \land\ ss = a\_block\_to\_sys\_spec\ ab2 \\
\qquad\quad \bullet\ pv \mapsto ss\} \\
\quad \Rightarrow a\_block\_to\_sys\_spec\ ab = SysSpec\ (\\
\qquad\qquad block\_specs \mathrel{\widehat{=}} bs, \\
\qquad\qquad z\_spec \mathrel{\widehat{=}} ass.subsys\_info.specification))
\end{array}
$$

7.2 Selecting Output

Note that in the following specification of output selection the number of passes over the *SYS_SPEC* is high because of the requirement that the order of output is determined by the order of occurrence of the selection patterns in the output filter specifications. The means that each pattern causes a new pass over the *SYS_SPEC*.

An extended path matching function is defined for selecting specifications for output. This function takes account of what has previously been output, and declines to output any paragraphs which match previous output paths.

z

$$\boldsymbol{output\_path\_match}: \mathbb{P}\ (PATH \times PATTERN \times \mathbb{P}\ PATTERN)$$

$$\begin{array}{l}
\forall\ c: PATH;\ bp: PATTERN\ ;\ pps: \mathbb{P}\ PATTERN\ \bullet\\
(c,\ bp,\ pps) \in output\_path\_match\\
\Leftrightarrow (c,bp) \in path\_match\\
\land\ (\forall p{:}pps \bullet\ (c,p) \notin path\_match)
\end{array}$$

This function is used for checking whether a path falls in the scope of a previously processed output filter specification. *sppe* and *sppi* are the excluded and included paths respectively, and the check is satisfied if the path does not match against the filter.

z

$$\boldsymbol{output\_path\_filter\_check}: \mathbb{P}\ (PATH \times (seq\ PATTERN) \times (seq\ PATTERN))$$

$$\begin{array}{l}
\forall\ c: PATH;\ sppe,\ sppi: seq\ PATTERN\ \bullet\\
(c,\ sppe,\ sppi) \in output\_path\_filter\_check\\
\Leftrightarrow (\forall p: ran\ sppi \bullet\ (c,\ p,\ ran\ sppe) \notin output\_path\_match)
\end{array}$$

This function checks a path against a complete sequence of output filter specifications. It only checks output for a particular output file so that duplication can be avoided in a single output file but allowed in distinct files. True if no match.

z

$$\begin{array}{l}
\boldsymbol{filter\_specs\_path\_check}:\\
\qquad \mathbb{P}\ (PATH \times STRING \times seq\ OUTPUT\_FILTER\_SPEC)
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\forall\ c: PATH;\ f: STRING;\ sofs: seq\ OUTPUT\_FILTER\_SPEC \bullet}\\
\multicolumn{2}{l}{(c,\ f,\ sofs) \in filter\_specs\_path\_check}\\
\Leftrightarrow & (\forall\ ofs: ran\ sofs\\
 & \mid f = ofs.output\_file\\
 & \bullet\ (c,\ ofs.excl,\ ofs.incl) \in output\_path\_filter\_check)
\end{array}$$

The following function filters the *SYS_SPEC* resulting from a model translation and extracts that part of the Z specification which defines the blocks matching a path (but not matching any previously output paths). This function takes one pass over the *SYS_SPEC* retrieving the specifications which match a single pattern (and have not been previously selected for this file).

z

$$
\begin{array}{|l}
\boldsymbol{select\_z}: (PATH \times PATTERN \times (\mathbb{P}\ PATTERN) \\
\qquad \times (seq\ OUTPUT\_FILTER\_SPEC) \times STRING) \\
\qquad \rightarrow SYS\_SPEC \rightarrow Z\_SPEC \\
\hline
\forall\ c: PATH;\ bp: PATTERN\ ;\ pps: \mathbb{P}\ PATTERN; \\
\quad sofs: seq\ OUTPUT\_FILTER\_SPEC;\ f: STRING;\ sys\_spec: SYS\_SPEC; \\
\quad block\_specs: BLOCK\_SPECS;\ z\_spec,\ sub\_z\_specs: Z\_SPEC \\
\mid sys\_spec = SysSpec\ (block\_specs \mathrel{\widehat{=}} block\_specs,\ z\_spec \mathrel{\widehat{=}} z\_spec) \\
\land\ sub\_z\_specs = {}^\frown/ \\
\qquad (\mu\ ssp{:}seq\ (seq\ Z\_PARA) \\
\qquad \mid ran\ ssp = ran \\
\qquad\quad (\lambda\ pv{:}PVALUE \bullet (select\_z\ (c \frown \langle pv \rangle,\ bp,\ pps,\ sofs,\ f)\ (block\_specs\ pv))) \\
\qquad\qquad\qquad) \\
\bullet\ select\_z\ (c,\ bp,\ pps,\ sofs,\ f)\ sys\_spec = \\
if\ (c,\ bp,\ pps) \in output\_path\_match \\
then \quad if\ (c,\ f,\ sofs) \in filter\_specs\_path\_check \\
\qquad then\ sub\_z\_specs \frown z\_spec \\
\qquad else\ sub\_z\_specs \\
else\ sub\_z\_specs
\end{array}
$$

The list version takes a list of paths for selection, shifting each path into the inhibit set after it has been used for selection.

z

$$
\begin{array}{|l}
\boldsymbol{list\_select\_z}: (PATH \times (seq\ PATTERN) \times (\mathbb{P}\ PATTERN) \\
\qquad \times (seq\ OUTPUT\_FILTER\_SPEC) \times STRING) \\
\qquad \rightarrow SYS\_SPEC \rightarrow Z\_SPEC \\
\hline
\forall\ c: PATH;\ bp: PATTERN;\ sbp: seq\ PATTERN; \\
\quad sofs: seq\ OUTPUT\_FILTER\_SPEC;\ f: STRING;\ pps: \mathbb{P}\ PATTERN; \\
\quad sys\_spec: SYS\_SPEC;\ block\_specs: BLOCK\_SPECS; \\
\quad z\_spec,\ sub\_z\_specs: Z\_SPEC \bullet \\
list\_select\_z\ (c,\ \langle\rangle,\ pps,\ sofs,\ f)\ sys\_spec = \langle\rangle \\
\land \quad list\_select\_z\ (c,\ \langle bp \rangle \frown sbp,\ pps,\ sofs,\ f)\ sys\_spec = \\
\quad (select\_z\ (c,\ bp,\ pps,\ sofs,\ f)\ sys\_spec) \\
\frown (list\_select\_z\ (c,\ sbp,\ pps \cup \{bp\},\ sofs,\ f)\ sys\_spec)
\end{array}
$$

The filter version takes a filter as a parameter,

z

$$
\begin{array}{l}
\boldsymbol{filter\_select\_z}: (PATH \times OUTPUT\_FILTER\_SPEC \\
\quad \times (seq\ OUTPUT\_FILTER\_SPEC) \times STRING) \\
\quad \rightarrow SYS\_SPEC \rightarrow Z\_SPEC \\
\hline
\forall\ c: PATH;\ ofs: OUTPUT\_FILTER\_SPEC;\ sofs: seq\ OUTPUT\_FILTER\_SPEC; \\
f: STRING;\ sys\_spec: SYS\_SPEC \bullet \\
filter\_select\_z\ (c,\ ofs,\ sofs,\ f)\ sys\_spec = \\
\quad if\ f = ofs.output\_file \\
\quad then\ (list\_select\_z\ (c,\ ofs.incl,\ ran\ ofs.excl,\ sofs,\ f)\ sys\_spec) \\
\quad else\ \langle\rangle
\end{array}
$$

Finally we specify how to compute all the output for a single output file from a sequence of output filter specifications. Two lists of output filter specifications are required, the first is a list of already processed specifications, the second a list of specifications not yet processed.

z

$$
\begin{array}{ll}
& \boldsymbol{list\_filter\_select\_z}: (PATH \times (seq\ OUTPUT\_FILTER\_SPEC) \\
& \quad \times (seq\ OUTPUT\_FILTER\_SPEC) \times STRING) \\
& \quad \rightarrow SYS\_SPEC \rightarrow Z\_SPEC \\
\hline
& \forall\ c: PATH;\ sofs1,\ sofs2: seq\ OUTPUT\_FILTER\_SPEC; \\
& \ ofs: OUTPUT\_FILTER\_SPEC;\ f: STRING;\ pps: \mathbb{P}\ PATTERN; \\
& \ sys\_spec: SYS\_SPEC \\
& \bullet \\
& list\_filter\_select\_z\ (c,\ sofs1,\ \langle\rangle,\ f)\ sys\_spec = \langle\rangle \\
\land & list\_filter\_select\_z\ (c,\ sofs1,\ \langle ofs\rangle \frown sofs2,\ f)\ sys\_spec = \\
& \quad (filter\_select\_z\ (c,\ ofs,\ sofs1,\ f)\ sys\_spec) \frown \\
& \quad (list\_filter\_select\_z\ (c,\ sofs1 \frown \langle ofs\rangle,\ sofs2,\ f)\ sys\_spec)
\end{array}
$$

7.3 Creating Output Files

The ClawZ translator will write selected parts of the resulting specification to one or more output files, provided that there have been no error reports or the flag *inhibit_output_on_error* is false. Note that these conditions for the output of the specifications are specified only informally, by this paragraph.

For each output file the following information is supplied as parameters to the ClawZ run:

1. The file name.
2. A list of paths for material whose output is to be inhibited.

3. A list of paths for material to be output.

The output phase of the ClawZ translator can be specified as follows:

z

$$\begin{array}{l}
\boldsymbol{create\_output}: SYS\_SPEC \rightarrow PVALUE \rightarrow \\
seq\ OUTPUT\_FILTER\_SPEC \rightarrow (STRING \twoheadrightarrow Z\_SPEC) \\
\hline
\forall\ sys\_spec:\ SYS\_SPEC;\ name: PVALUE; \\
sos: seq\ OUTPUT\_FILTER\_SPEC \bullet \\
create\_output\ sys\_spec\ name\ sos = \\
\{output\_file: STRING;\ excl, incl: seq\ PATTERN;\ spec{:}Z\_SPEC \\
\mid (output\_file \mathrel{\widehat{=}} output\_file,\ excl \mathrel{\widehat{=}} excl,\ incl \mathrel{\widehat{=}} incl) \in ran\ sos \\
\land\ spec = list\_filter\_select\_z\ (\langle name \rangle, \langle\rangle, sos, output\_file)\ sys\_spec \\
\bullet\ output\_file \mapsto spec\}
\end{array}$$

The result is a mapping from filenames to Z_SPECs.

8 RUNNING CLAWZ

First a procedure for processing artificial subsystems:

z

$$\begin{array}{l}
\boldsymbol{proc\_art\_subsys}: META\_FILE \rightarrow BOOL \\
\quad \rightarrow (A\_BLOCK \times ART\_SUBSYS\_SPEC) \\
\quad \rightarrow (STRING \twoheadrightarrow Z\_SPEC) \times META\_FILE \\
\hline
\forall\ ab, ab':\ A\_BLOCK;\ ass: ART\_SUBSYS\_SPEC;\ ss: SYS\_SPEC; \\
\quad lib: BOOL;\ output: STRING \twoheadrightarrow Z\_SPEC;\ imf, mf: META\_FILE \\
\mid ab \mapsto (ab', mf) \in (make\_artificial\_subsys\_block\ imf\ lib\ ass) \\
\land\ ss = a\_block\_to\_sys\_spec\ ab' \\
\land\ output = create\_output\ ss\ ass.name\ ass.output\_spec \\
\bullet\ proc\_art\_subsys\ imf\ lib\ (ab, ass) = (output, mf)
\end{array}$$

We now tie the whole thing together with a top level specification showing how a collection of files are created from the *SYSTEM* derived from the Simulink model as indicated in the *RUN_PARAMS*. The following specification gives a sequence of maps from filenames to specifications. It is to be understood that if the same filename occurs in the domain at more than one place in this sequence then the relevant specifications will be written to that file in the order in which they occur in the sequence.

The synthesize_blocks function returns the set of paths for blocks which have not been translated. This set should be printed in a single warning, in which the paths are ordered and printed in an indented column one to a line.

A *META_FILE*, is also to be output if a filename has been supplied for it. Note that this metadata is generated too late to be used for block references internal to a library, and should be supplied to clawz in a subsequent run of clawz if such references are present.

z

$$
\begin{array}{l}
\boldsymbol{clawz\_run}: (SYSTEM \times META\_FILE \times STRUCTURE \times RUN\_PARAMS) \\
\quad \rightarrow (seq\ (STRING \mathrel{+\!\!\!\!\!\twoheadrightarrow} Z\_SPEC)) \times (OPT[STRING] \times META\_FILE) \\
\hline
\forall \quad lib: BOOL;\ sys: SYSTEM; \\
\qquad sys\_c{:}[pars: \mathbb{F}\ PARAM;\ blocks: \mathbb{F}_1\ BLOCK;\ lines: \mathbb{F}_1\ LINE]; \\
\qquad mf,\ mflib: META\_FILE;\ clawz\_params: RUN\_PARAMS; \\
\qquad ab1,\ ab2,\ ab3,\ ab4,\ ab5,\ ab6,\ vs: A\_BLOCK;\ ss: SYS\_SPEC; \\
\qquad f1: STRING \mathrel{+\!\!\!\!\!\twoheadrightarrow} Z\_SPEC;\ system\_name: PVALUE;\ failpaths: \mathbb{F}\ PATH; \\
\qquad ssfs: seq((STRING \mathrel{+\!\!\!\!\!\twoheadrightarrow} Z\_SPEC) \times META\_FILE); \\
\qquad steerfile: STRUCTURE;\ mvartypes: MVARTYPES \\
\mid lib = (clawz\_params.meta\_output \neq Nil) \\
\land System\ sys\_c = sys \\
\land mvartypes = mvts\_of\_structure\ steerfile \\
\land (name \mathrel{\widehat=} Name,\ value \mathrel{\widehat=} system\_name) \in sys\_c.pars \\
\land ab1 = system\_to\_a\_block\ lib\ system\_name\ sys \\
\land ab2 = library\_lookup\ mf\ ab1 \\
\land ab3 = set\_port\_types\ ab2 \\
\land ab4 = propagate\_signal\_details\ mvartypes\ ab3 \\
\land vs = if\ virtualize\ then\ virtualize\_system\ mvartypes\ ab4\ else\ ab4 \\
\land (failpaths,\ ab5) = synthesize\_blocks\ mvartypes\ vs \\
\land ab6 = translate\_system\ ab5 \\
\land mflib = if\ lib\ then\ extract\_metadata\ (NullString,\ \langle system\_name\rangle)\ ab6\ else\ \langle\rangle \\
\land ss = a\_block\_to\_sys\_spec\ ab6 \\
\land f1 = create\_output\ ss\ system\_name\ clawz\_params.output\_spec \\
\land ssfs = \quad \{n: \mathbb{N};\ asss: ART\_SUBSYS\_SPEC; \\
\qquad f: STRING \mathrel{+\!\!\!\!\!\twoheadrightarrow} Z\_SPEC;\ mf: META\_FILE \\
\qquad \mid n \mapsto asss \in clawz\_params.art\_subsys\_specs; \\
\qquad\ \ (ab6,\ asss) \mapsto (f,\ mf) \in proc\_art\_subsys\ mf\ lib \\
\qquad \bullet\ n \mapsto (f,\ mf)\} \\
\bullet \\
clawz\_run\ (sys,\ mf,\ steerfile,\ clawz\_params) = \\
(\langle f1\rangle \frown (ssfs \mathbin{\text{⨾}} first),\ (clawz\_params.meta\_output,\ \frown/(\langle mflib\rangle \frown (ssfs \mathbin{\text{⨾}} second))))
\end{array}
$$

9 ENHANCING CLAWZ

In this section are presented notes on how to go about enhancing this specification.

9.1 Adding A Parameter Translation Method

When new blocks are added to the library it often occurs that some new kind of parameter is required by the block which cannot be translated into Z by any of the available methods. It then becomes necessary to resort to specifying individually each instance of the block in the libarary, and selecting the appropriate instance using the untranslatable parameter.

Here we note the aspects of the specification which must be upgrading to provide support for translating a new parameter into Z so that a single library specification can be used for a range of values of the parameter. No account here is taken of the work necesary to synthesize or virtualize blocks with the new kind of parameter.

The aspects of the specification which need to be adjusted are as follows.

2.6 Add additional clauses to the Parameter Translation Code Grammar as required.

2.7 Specify the concrete syntax of the new parameters.

3.4 Specify the datatype corresponding to the abstract syntax for each new parameter translation type.

3.6 Add declarations for any new constants which will be used in the Z translation of the parameters. Extend the abstract syntax of Z if necessary for the translations.

3.7 Define a new $PVALUE$ for the new translation code.

4 This is where complications are likely to arise.

The obvious points of change are mentioned below, but depending on exactly what the new parameter type is more extensive changes might prove necessary.

4.2 Add a declaration for the function which parses according to the new grammar.

4.6 Upgrade this section to translate the parsed parameter into a Z expression.

10 INDEX