# Contents

1. Overview ......................................................................................................................... 3  
2. Methodology ................................................................................................................... 4  
3. The language used to write business rules ...................................................................... 5  
   3.1 Structure of a function body. .................................................................................... 5  
   3.3 The assignment statement. ........................................................................................ 5  
   3.3 The IF-ELSE-ENDIF statement ............................................................................... 5  
   3.4 The WHILE-DO-LOOP statement ........................................................................... 6  
   3.4 The FOR-DO-LOOP statement ................................................................................ 6  
   3.5 The official syntax BNF ............................................................................................ 6  
4. Local variables and argument names .............................................................................. 9  
5. Comments ..................................................................................................................... 10  
6. Built in “native” functions ............................................................................................ 11  
7. Text ............................................................................................................................... 13  
8. Programmer’s guide to SBR ..................................................................................... 14  
   8.1 Prototyping a business rule function ....................................................................... 14  
   8.2 Using the “load” method to read in the “.sbr” files. ............................................... 15  
   8.3 Passing data to and from the SBR functions .......................................................... 15  
   8.4 Invoking a function at runtime ............................................................................... 16  
   8.5 Application defined native functions ...................................................................... 17  
   8.6 Prototyping a native function .................................................................................. 18  
   8.7 Advanced text output .............................................................................................. 19  
   8.8 Complete Example ................................................................................................. 20
1. Overview

Interpreted business rules provides a mechanism for the users of a simulation model to modify some of the code in the model that is used at crucial decision points. This code can be modified without having to recompile the model. “Functions” are written in a simplified language called SBR or (SIMSCRIPT Business rules). Each interpreted SBR function is given a name and is called from within the SIMSCRIPT model using that name. The SBR function will accept a list of arguments and must return with an integer number, real number, text or pointer value. For example, the following SBR function will return a pointer to a supply site.

```plaintext
function getSupplySiteWithLowestShelfDays(site, part)
    minDays = rinf_c
    supplySite = getFirstSupplySite
    while supplySite <> null
        do
            days = getShelfDays(site, part) + random_f(1, 0.0, 1.0)
            if days < minDays
                minDays = days
                bestSite = supplySite
            endif
            supplySite = getNextSupplySite(supplySite)
        loop
    return bestSite
end
```

SBR functions can call each other, can make recursive calls, or can call native functions that are implemented within the SIMSCRIPT III model. “Built-in” native functions are provided (such as the call to “random_f” above).
2. Methodology

The SIMSCRIPT Business Rules Interpreter is written using a recursive descent parsing algorithm. The “code” for the business rules is kept in one or more flat (ascii) files and is loaded and parsed once on initialization.

The parsing algorithm will construct a hierarchical parse-tree data structure. This is essentially an intermediate representation of the code that can be “executed” at the time a business rule is invoked. Errors in syntax or semantics will be printed at the time the business rules are loaded. If a syntax error is detected, the SBR function being loaded cannot be executed. Runtime errors may also result during actual execution. For example, if an attempt is made to divide by zero, a runtime error will be printed and execution of the function will stop immediately.
3. The language used to write business rules

The language used for the SIMSCRIPT business rules will be subsequently referred to as SBR.

3.1 Structure of a function body

A function must begin with the “function” keyword and be terminated with the “end” keyword. The name of the function follows the “function” keyword. An argument list may be included after the name. Within the body of the function, there must be at least one “return” statement.

```plaintext
function getMax(i1, i2)
    return imax_f(i1,i2)
end
```

All SBR functions must be explicitly prototyped in the SIMSCRIPT III source code. (Prototyping is described later). Function prototyping establishes the “mode” of all arguments to the function and the return mode itself. An error message will be generated on initialization if an unprototyped function is encountered as a result of the “load” statement.

3.3 The assignment statement

The “=” is used as an assignment operator. The name of a local variable must appear on the left, and an expression on the right. It is not legal to assign a pointer to anything else but another pointer. (To set the value of a pointer variable to zero the value “null” can be used.) Integers and real values are assignment compatible. When a real value is assigned to an integer variable, the value is rounded.

```plaintext
myValue = b + sqrt_f(b * b - 4 * a * c) / (2 * a)
```

Text can be assigned to another text variable or a quoted string. The “+” operator can be used to concatenate text strings.

```plaintext
myTextVariable = "hello" + " " + "world"
```

3.3 The IF-ELSE-ENDIF statement

Conditionals statements can be written in SBR. The “and”, “or” and “not” keywords are used as Boolean logic operators while the “=”, “<>”, “<”, “>” ”, “<=” and “>=” are used for comparison. When testing pointer variables for zero, the keyword “null” should be used instead of “0”.

```plaintext
myTextVariable = "hello" + " " + "world"
```
if batch > 0 and netRequisitions >= batch
    batchCount = netRequisitions / batch
else
    batchCount = 0
endif

3.4 The WHILE-DO-LOOP statement

Loops based on conditionals are implemented using the “WHILE” statement. Code to be executed must come between the “DO” and “LOOP” keywords. Nested loops are allowed.

supplySite = getFirstSupplySite
while supplySite <> 0
do
    numSites = numSites + 1
    supplySite = getNextSupplySite(supplySite)
loop

3.4 The FOR-DO-LOOP statement

The classic “for” loop is supported. An iterator is initially assigned to the given “low” value and incremented until it reaches the value specified after the “to” clause. An optional “by” clause allows you to specify a value to be added to the iterator at the top of the loop.

minHeight = rinf_c
for i = 1 to 100 by 4
do
    let y = getHeight(getBoxIndex(box(i)))
    minHeight = min_f(minHeight, y)
loop

3.5 The official syntax BNF

The following notation will be used:

The “<>” will be used to enclose a non-terminal. For example: < non-terminal >.

The quote marks will enclose a keyword that is part of the language. For example: “terminal”.
The braces “{“ and “}” will enclose optional syntax that can be repeated zero or more times. For example:  
{ “,” <name> }  

The brackets “[“ and “]” will enclose optional syntax that can be repeated only once. For example [ <namelist> ]  

An identifier in the language can be any sequence of alphanumeric characters beginning with a letter. The underscore “_” can also be used in a name. Names can be upper, lower or mixed case, but the language is case-insensitive.  

In the rules, the identifier is represented with “< name >”.  

SBR is represented by the following syntax rules:  

```
< business_rule >  => < heading > { <statement> } “end”
< heading >  => “function” < name > [ <namelist> ]
< namelist >   => “(“ <name> { “,” <name> } “)“)
< statement >   => < assignment_stmt >  
                => < while_stmt >  
                => < if_stmt >
< assignment_stmt > => <name> “=” < expression >
<while_stmt >  => “while” <bool_expression> “do” { < statement >… } “loop”
<for_stmt >   => “for” <name> “=” < expression > “to” < expression >  
               [ “by” < expression > ] “do” { < statement >… } “loop”
< if_stmt >  => “if” <bool_expression> { <statement> … }  
               [ “else” { <statement> } ] “endif”
< return_stmt >  => “return” <expression>
< expression >  => [ < add_op > ] < term > { < add_op> < term >… }  
< add_op >   => “+”  
               => “-”  
< term >  => < factor > { < mul_op > < factor > }
< factor >   => < primary > [ <primary_op> < primary > ]
< primary >   => < literal >  
               => “(“ < expression > “)“
< mul_op >   => “*”  
               => “/”
< primary_op >   => “%”  
                => “**”
< literal >   => < integer_constant >  
               => < real_constant >  
               => < string_constant >  
               => “null”
               => < name > { <expression_list> }
< expression_list > => “(“ < expression > { “,” < expression > }… “)”
```
<bool_expression> => <relation> { <logical_op> <relation>… }

<logical_op> => “and”
=> “or”

<relation> => <expression> <rel_op> <expression>
=> “(<bool_expression> “)”
=> “not” “(<bool_expression> “)”

<rel_op> => “>”
=> “<”
=> “=”
=> “>=”
=> “<=”
4. Local variables and argument names

In SBR, locally used variables are not declared explicitly. The modes are determined empirically, during an assignment statement. It is therefore not allowed to use a local variable that has not been assigned a value. To determine the mode of an expression used on the right hand side of an expression, the modes of constants and native function calls are used.
5. Comments

Comments can be placed within the business rule code. They are delimited with the “/~” and “~/” sequences.

/~ Here is a comment on a
couple of lines ~/ 
function rTestFunction ( i1, i2 )
   return with i2  ~/ here is another comment ~/ 
end
6. Built in native functions

A function written in SBR can directly invoke functions written in SIMSCRIPT III. These are called "native" functions. There are numerous "built in" native functions that are provided to perform mathematical or other operations. Some of the native functions will return SIMSCRIPT III predefined constants, or SIMSCRIPT global variables (like TIME.V). The following functions are provided:

<table>
<thead>
<tr>
<th>Name</th>
<th>Mode</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp_c</td>
<td>real</td>
<td>&lt;none&gt;</td>
<td>Constant for &quot;e&quot;.</td>
</tr>
<tr>
<td>inf_c</td>
<td>int</td>
<td>&lt;none&gt;</td>
<td>Maximum value for a 32-bit integer</td>
</tr>
<tr>
<td>rinf_c</td>
<td>real</td>
<td>&lt;none&gt;</td>
<td>Maximum value for a 64-bit real</td>
</tr>
<tr>
<td>pi_c</td>
<td>real</td>
<td>&lt;none&gt;</td>
<td>Pi constant.</td>
</tr>
<tr>
<td>radian_c</td>
<td>real</td>
<td>&lt;none&gt;</td>
<td>Converts degrees to radians 180 / pi</td>
</tr>
<tr>
<td>abs_f</td>
<td>real</td>
<td>r</td>
<td>Returns the absolute value of argument 1</td>
</tr>
<tr>
<td>iabs_f</td>
<td>int</td>
<td>i</td>
<td>Returns the absolute value of argument 1</td>
</tr>
<tr>
<td>arccos_f</td>
<td>real</td>
<td>r</td>
<td>Returns the arc cosine of argument 1</td>
</tr>
<tr>
<td>arcsin_f</td>
<td>real</td>
<td>r</td>
<td>Returns the arc sine of argument 1</td>
</tr>
<tr>
<td>arctan_f</td>
<td>real</td>
<td>r,r</td>
<td>Returns the arc tangent of arg2 / arg1</td>
</tr>
<tr>
<td>beta_f</td>
<td>real</td>
<td>r,r,i</td>
<td>Calls the SIMSCRIPT III beta.f function.</td>
</tr>
<tr>
<td>binomial_f</td>
<td>real</td>
<td>i,r,i</td>
<td>Calls the SIMSCRIPT III binomial.f function.</td>
</tr>
<tr>
<td>cos_f</td>
<td>real</td>
<td>r</td>
<td>Returns the cosine of argument 1</td>
</tr>
<tr>
<td>erlang_f</td>
<td>real</td>
<td>r,i,i</td>
<td>Calls the SIMSCRIPT III erlang_f function.</td>
</tr>
<tr>
<td>exp_f</td>
<td>real</td>
<td>r</td>
<td>&quot;e&quot; to the arg</td>
</tr>
<tr>
<td>exponential_f</td>
<td>real</td>
<td>r,i</td>
<td>Calls the SIMSCRIPT III exponential.f function.</td>
</tr>
<tr>
<td>frac_f</td>
<td>real</td>
<td>r</td>
<td>Returns the fractional part of a real number.</td>
</tr>
<tr>
<td>gamma_f</td>
<td>real</td>
<td>r,r,i</td>
<td>Calls the SIMSCRIPT III gamma.f function.</td>
</tr>
<tr>
<td>length_f</td>
<td>real</td>
<td>t</td>
<td>Returns the length of a text value.</td>
</tr>
<tr>
<td>itot_f</td>
<td>text</td>
<td>i</td>
<td>Returns the text representation of an integer.</td>
</tr>
<tr>
<td>log_e_f</td>
<td>real</td>
<td>r</td>
<td>Return the log base “e” of arg 1</td>
</tr>
<tr>
<td>log_normal_f</td>
<td>real</td>
<td>r,r,i</td>
<td>Calls the SIMSCRIPT III log.normal.f function.</td>
</tr>
<tr>
<td>log_10_f</td>
<td>real</td>
<td>r</td>
<td>Returns log base 10 of arg 1</td>
</tr>
<tr>
<td>lower_f</td>
<td>text</td>
<td>t</td>
<td>Converts the given text to lower case</td>
</tr>
<tr>
<td>match_f</td>
<td>text</td>
<td>t,t,i</td>
<td>Calls the SIMSCRIPT III match.f function</td>
</tr>
<tr>
<td>min_f</td>
<td>real</td>
<td>r,r,r,…</td>
<td>Returns the minimum of several arguments.</td>
</tr>
<tr>
<td>min_f</td>
<td>integer</td>
<td>i,i,…</td>
<td>Returns the minimum of several arguments.</td>
</tr>
<tr>
<td>imax_f</td>
<td>real</td>
<td>r,r,r,…</td>
<td>Returns the maximum of several arguments.</td>
</tr>
<tr>
<td>imax_f</td>
<td>integer</td>
<td>i,i,…</td>
<td>Returns the maximum of several arguments.</td>
</tr>
<tr>
<td>normal_f</td>
<td>real</td>
<td>r,r,i</td>
<td>Calls the SIMSCRIPT III normal.f function.</td>
</tr>
<tr>
<td>poisson_f</td>
<td>real</td>
<td>r,i</td>
<td>Calls the SIMSCRIPT III poisson function.</td>
</tr>
<tr>
<td>process_v</td>
<td>pointer</td>
<td>&lt;none&gt;</td>
<td>Returns the current process notice.</td>
</tr>
<tr>
<td>randi_f</td>
<td>integer</td>
<td>i,i,i</td>
<td>Calls the SIMSCRIPT III randi.f function.</td>
</tr>
<tr>
<td>random_f</td>
<td>real</td>
<td>i</td>
<td>Returns a random number between 0 and 1.</td>
</tr>
<tr>
<td>Function</td>
<td>Type</td>
<td>Arguments</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td>-----------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td><code>repeat_f</code></td>
<td>text</td>
<td>t,i</td>
<td>Repeats the given text string arg2 times</td>
</tr>
<tr>
<td><code>rtot_f</code></td>
<td>text</td>
<td>r,i,i,i</td>
<td>Real to text given fw, precision, use_exp</td>
</tr>
<tr>
<td><code>sign_f</code></td>
<td>integer</td>
<td>r</td>
<td>-1 for neg, 0 for zero, 1 for pos</td>
</tr>
<tr>
<td><code>rsin_f</code></td>
<td>real</td>
<td>r</td>
<td>Returns the sine of arg 1.</td>
</tr>
<tr>
<td><code>rsqrt_f</code></td>
<td>real</td>
<td>r</td>
<td>Returns the square root of arg1.</td>
</tr>
<tr>
<td><code>time_v</code></td>
<td>real</td>
<td>&lt;none&gt;</td>
<td>Returns time.v, the current simulation time.</td>
</tr>
<tr>
<td><code>triang_f</code></td>
<td>real</td>
<td>r,r,r,i</td>
<td>Calls the SIMSCRIPT III triang.f function</td>
</tr>
<tr>
<td><code>trim_f</code></td>
<td>text</td>
<td>t,i</td>
<td>Removes leading and/or trailing blanks.</td>
</tr>
<tr>
<td><code>trunc_f</code></td>
<td>integer</td>
<td>r</td>
<td>Returns the truncated value of arg 1</td>
</tr>
<tr>
<td><code>uniform_f</code></td>
<td>real</td>
<td>r,r,i</td>
<td>Calls the SIMSCRIPT III uniform.f function.</td>
</tr>
<tr>
<td><code>upper_f</code></td>
<td>text</td>
<td>t</td>
<td>Converts arg1 to upper case.</td>
</tr>
<tr>
<td><code>weibull_f</code></td>
<td>real</td>
<td>r,r,i</td>
<td>Calls the SIMSCRIPT III weibull.f function.</td>
</tr>
<tr>
<td><code>length_f</code></td>
<td>integer</td>
<td>t</td>
<td>Returns the length of a text value.</td>
</tr>
</tbody>
</table>
7. Text

In SBR text variables can be assigned and passed as arguments. If used in expressions where the operators are “+”, the text will be concatenated.

```plaintext
ts = " ";
t1 = "hello";
t2 = "there";
t3 = "world";
tMyText = t1 + ts + t2 + ts + t3
```

The following native functions can be used for text manipulation.

<table>
<thead>
<tr>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>length_f(tval)</td>
<td>Returns the length of the text string</td>
</tr>
<tr>
<td>itot_f(ival)</td>
<td>Converts an integer to text</td>
</tr>
<tr>
<td>rtot_f(ival, field_width, precision, use_exp)</td>
<td>Convert real value to text representation.</td>
</tr>
<tr>
<td>tower_f(tval)</td>
<td>Convert to lower case</td>
</tr>
<tr>
<td>upper_f(tval)</td>
<td>Convert to upper case</td>
</tr>
<tr>
<td>tsubstr_f(tval, pos, length)</td>
<td>Returns the substring starting at position “pos”. “length” characters are copied.</td>
</tr>
<tr>
<td>match_f(tval, pattern)</td>
<td>Returns the position within tval of the first occurrence of “pattern”, or zero if there is no such occurrence; the search begins after skipping the first “offset” characters of tval.</td>
</tr>
<tr>
<td>trim_f(tval, flag)</td>
<td>Returns a copy of “tval” which has leading and/or trailing blanks removed; if “flag” 0, leading blanks are removed; if “flag” ≥ 0, trailing blanks are removed. To remove both leading and trailing blanks, use “flag” = 0.</td>
</tr>
</tbody>
</table>


8. Programmer’s guide to SBR

To implement a model using the SBR interpreter, the preamble must import from the sbr subsystem found in sbr.m.sim. The interpreter class provides the basis for using SBR. This class should be sub-classed if the existing native functionality is to be extended (which is usually the case).

Preamble for the my_model subsystem including the sbr.m subsystem
begin class my_interpreter
  every my_interpreter is an interpreter and
  overrides the native, "if SIMSCRIPT III fns defined
  overrides the prototype "always needed
end

define the_interpreter as a my_interpreter reference variable
end

8.1 Prototyping a business rule function

In order to insure that the SBR interpreter can interact properly with the business rule code, all business rule functions must be prototyped by overriding the prototype method of the interpreter class in the public or private preamble.

define prototype as a text "virtual" method given
  1 text argument "name of native function, or rule

  "returns with a prototype text for the arguments of the
  "named native function or business rule. The first char in the text
  "represents the return mode while each subsequent char in the text
  "represents the mode of an argument. The codes are:
  ""i" = integer, "r" = real, "p" = pointer and "t" = text.
  "The "+" character can be used to indicate the previous arg
  "repeats.
  "If the function is not defined, return with "".

This method is called when the interpreter reads the heading for a business rule in the SBR code. The name of the function is passed to the prototype method. The code written for prototype can then compare the name with the list of known functions (using a “select case” statement). If the name is found, an encoded text string is returned by the prototype method which describes both the return type and given arguments. The interpreter will compare the number of arguments in the heading of the business rule with the number of characters in the prototype text.

Each character in the text string returned by the prototype method represents a “mode”. The first character is the return mode, followed by given argument modes. (Therefore the prototype text string for a business rule with “n” arguments will be “n+1” in length.) The character codes are shown below.
<table>
<thead>
<tr>
<th>Mode character</th>
<th>SIMSCRIPT III mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>INTEGER</td>
</tr>
<tr>
<td>r</td>
<td>REAL</td>
</tr>
<tr>
<td>t</td>
<td>TEXT</td>
</tr>
<tr>
<td>p</td>
<td>POINTER</td>
</tr>
</tbody>
</table>

For example, the prototype text for a function that accepts an INTEGER, TEXT, and POINTER (in that order) and returns a REAL would be “ritp”.

NOTE: The prototype method should always return with “interpreter'prototype(name)” if the given name is not defined by the application. This will ensure that calls to built-in native functions will operate properly.

### 8.2 Using the load method to read in the .sbr files.

Reading in the file containing the SBR code is accomplished using the load method of the Interpreter class. This method takes the name of the code file as its argument and returns “1” if the compilation is successful, “0” otherwise. The load method can be called many times to load different file. For example, to load in the file “SEM.sbr”:

```python
create the_interpreter
call load(the_interpreter)("SEM.sbr")
```

The load method will parse the code in the “.sbr” file and build the intermediate (internal) representation. The method will return with “0” if any syntax or semantics errors are encountered in any of the functions.

### 8.3 Passing data to and from the SBR functions.

The SBR functions will usually be designed with arguments of various modes. Also, each function will return with a data value that must be read in by the calling code. The variable class found in the sbr.m subsystem is used to hold various representations of data values. Instances of variable objects are passed as arguments to SBR functions. Another variable instance is passed to the function to hold the return value.

The “data” being stored in the variable can have one of four modes. The following constants define the modes

<table>
<thead>
<tr>
<th>MODE</th>
<th>DATA</th>
<th>PROTOTYPE CHAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer_mode</td>
<td>32-bit signed integer.</td>
<td>i</td>
</tr>
<tr>
<td>real_mode</td>
<td>64-bit floating point</td>
<td>r</td>
</tr>
<tr>
<td>text_mode</td>
<td>Text string.</td>
<td>t</td>
</tr>
<tr>
<td>pointer</td>
<td>32-bit pointer.</td>
<td>p</td>
</tr>
</tbody>
</table>
The following methods are used to “set” and “get” the values in a variable instance:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_integer</td>
<td>returns the integer representation of the data. If the mode of the data is _real_mode, the value is rounded. Text data will be automatically converted for textual integers.</td>
</tr>
<tr>
<td>get_real</td>
<td>returns the real representation of the data.</td>
</tr>
<tr>
<td>get_text</td>
<td>returns the real representation of the data.</td>
</tr>
<tr>
<td>get_pointer</td>
<td>Return with the pointer.</td>
</tr>
<tr>
<td>set_integer</td>
<td>Sets the data given an integer.</td>
</tr>
<tr>
<td>set_real</td>
<td>Sets the data given a real.</td>
</tr>
<tr>
<td>set_text</td>
<td>Sets the data as text.</td>
</tr>
<tr>
<td>set_pointer</td>
<td>Set the data as a pointer.</td>
</tr>
</tbody>
</table>

### 8.4 Invoking a function at runtime.

Once functions in the file have been loaded they can be executed. The interpreter `evaluate` method will execute a business rule function:

```python
define evaluate as a integer method given 1 text argument, 1 variable reference argument, 1 1-dim variable reference argument  
"Calls and evaluates an existing business rule."
```

The evaluate method returns “1” if the rule has been performed successfully, “0” otherwise. The name of the rule is given which must match the name used in the function heading in the SBR file. A variable reference must be allocated and passed as the second argument. The third argument is an array of variable references that contain the arguments to the function. If there are no arguments to the SBR function, this argument can be zero.

There are two variable class methods that can be used to reserve, allocate, and deallocate the argument list array: `create_args()` and `destroy_args()`. Once the argument list has been created, the elements of the array can be assigned by calling the “set_integer”, “set_real”, etc. methods.

For example, suppose we want to call the SBR function listed in (1) i.e:

```python
function pSupplySiteWithLowestShelfDays(pSite, pPart)
```

The function need two pointer variables as arguments and returns a pointer. We must first create the variable that is to contain the result, then reserve the array representing the argument list. The `evaluate` method is called to execute the rule. The `result` variable can
then be queried to get the data value returned from the SBR function. The argument list array and result variable can then be deallocated.

```plaintext
define arglist as a 1-dim Variable reference variable
define result_var as a Variable reference variable

create result_var
let arglist = variable'create_args(2)
call set_pointer(arglist(1))(theCurrentSite)
call set_pointer(arglist(2))(theCurrentPart)

if evaluate(the_interpreter)("pSupplySiteWithLowestShelfDays", result_var, arglist) <> 0
  let theLowestShelfDaySite = get_pointer(result_var)
always

call Variable'destroy_args(arglist)
destroy this result_var
```

### 8.5 Application defined native functions

“Native” functions are written in SIMSCRIPT III and compiled with the rest of the model code. As was mentioned previously, there are many native “built in” SIMSCRIPT functions that can be called from the SBR code. When the interpreter reads a name that has not yet been assigned a value locally, it calls its own native method to see if the reference is to a SIMSCRIPT function. Internally the native method compares this name against the list of known names. If the name matches one of the built in functions, the function is executed, and “1” is returned. If no matching function is found, “0” is returned. The method is defined below:

```plaintext
define native as a integer method given
  1 text argument,                      "name of native function
  1 variable reference argument,       "store the result here
  1 1-dim variable reference argument  "arguments will be here
  "Override this method to allow the business rule to call a
  "native simscript function. If zero is returned, the interpreter
  "will assume the function is not-implemented and will try to invoke
  "one of the built in native functions. If that fails a runtime
  "error is generated.
```

The native method takes the same arguments as the evaluate method. These include the function name, a Variable reference that is to hold the return value, and an array of Variables for the argument list. The SIMSCRIPT application can override this method and compare the name against its own list of names. If a match is not found, the “inherited” native method can be called to try for a match. If a match is found, the array of Variable references (argument list) can be unpacked and the SIMSCRIPT function called. (There is no need to allocate the result variable or argument list array, they are allocated by the caller).
Additional native functions can be defined specifically for the SEM application. These “add on” function are necessary to provide a way for the implemented business rules to communicate with the simulation. For example, here are some possible SEM native functions that can be invoked in the business rule.

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getFirstSupplySite</td>
<td>&lt;none&gt;</td>
<td>Returns first site</td>
</tr>
<tr>
<td>getNextSupplySite</td>
<td>pSite</td>
<td>Returns the next site in the list.</td>
</tr>
<tr>
<td>backOrderDays</td>
<td>pSite, pPart</td>
<td>Returns # back order days for a given part at given site</td>
</tr>
<tr>
<td>globalCondemns</td>
<td>pPart</td>
<td>Returns the number of condemns for a part</td>
</tr>
<tr>
<td>globalBuildRequisitions</td>
<td>pPart</td>
<td>Returns # of build requisitions</td>
</tr>
</tbody>
</table>

Part of the SEM derived interpreter can be written as follows:

```plaintext
method sem_interpreter'native(name, result_var, arglist)
  define partPtr as a PartObj reference variable
  define sitePtr as a SiteObj reference variable

  select case name
    case "globalPartsBuilt"  ''return the GlobalPartsBuilt attribute
      partPtr = get_pointer(arglist(1))
      call set_integer(result_var)(globalPartsBuilt(partPtr))
    case "globalCondemns"  ''return the GlobalCondemns attribute
      partPtr = get_pointer(arglist(1))
      call set_integer(result_var)(globalCondemns(partPtr))
    case "shelfDays"  ''computes the # shelf dat attribute
      partPtr = get_pointer(arglist(1))
      sitePtr = get_pointer(arglist(2))
      call set_integer(result_var)(getShelfDays(partPtr)(sitePtr))
    default
      return with Interpreter'native(name, result_var, arglist)
  endselect

  return with 1
end

8.6 Prototyping a native function

In order to insure that the SBR interpreter can interact properly with the business rule code, all native functions must not only be implemented by overriding the ‘native’ method, and adding code to unpack arguments etc, .. but must also be prototyped via the prototype method described in Section 8.1.

When the interpreter reads a function call in the SBR code it will check to make sure that a prototype exists for the function. The name of the function is passed to the prototype
method. The code written for prototype can then compare the name with the list of known functions (using a “select case” statement). If the name is found, an encoded text string is returned which describes both the return type and given arguments.

Native functions can take repeat arguments, but must be prototyped as such. If the last argument is repeated, a “+” must be appended to the end of the prototype text. For example, the prototype text for a function that returns with an INTEGER but accepts one or more REAL arguments would be “ir+”.

### 8.7 Advanced text output

By default output from the write statement as well as error messages will be printed to “standard output” (console window, unit 6) while error messages are printed to “standard error” (console window, unit 98). There may be cases where text output needs to be specially handled. (For example, to have the message printed to a file or a popup dialog box.) This can be accomplished by overriding the output method of the Interpreter class.

```plaintext
define output as a method given
  1 text argument,         ''text to be printed out
  1 integer argument       ''reason:
    ''_output_write -> write statement
    ''_output_write_crlf->write with return
    ''_output_compile_error
    ''_output_runtime_error

  ''This method is called when the Interpreter needs to print text. This
  ''will occur if a "write" statement is executed or if an error is
  ''encountered while reading or executing SBR code. The reason for
  ''printing is passed as the second argument. By default the text
  ''message will printed to standard output. This can be overridden
  ''to provide customized printing of messages.

The output method is called by the interpreter whenever text is to be printed as the result of a read (compile) error, a runtime error or by executing a “print” statement. The first argument contains the text to print out while the second argument (cause) specifies why the message is being printed. This may be one of the following values:

- `_output_write` The flush character (+) was found in a write statement
- `_output_write_crlf` The carriage return character (/) was encountered while evaluating a write statement
- `_output_compile_error` Error message for a compile error discovered while loading SBR code.
- `_output_runtime_error` Error message for a runtime error.

If the output method is not overridden, error output will be printed to “standard error” and text output will go to “standard output”. Note that when using the “write” statement the output method is only called at the time a newline “/” or flush “+” is written.
The following example shows how to print text messages from the “write” statement to the file “sbr_output.txt” and error messages to the file “sbr_errors.txt”.

''example code: overriding the output method to handle text output differently
method my_interpreter'output(message, cause)
  define files_opened as an integer saved variable

  if files_opened = 0
    let files_opened = 1
    open unit 1 for output, name is "sbr_output.txt"
    open unit 2 for output, name is "sbr_errors.txt"
  always

  select case cause
    case _output_write
      write message as T * using 1  ''print to file
    case _output_write_crlf
      write message as T *, / using 1  ''print to file
    case _output_compile_error, _output_runtime_error
      write message as T *, / using 2  ''print to file
  endselect
end

8.8 Complete Example

A complete example of the SIMSCRIPT Business Rules Interpreter can be found in the SIMSCRIPT III examples folder distributed with the SIMSCRIPT III product at:

$SIMHOME/sim_examples/sim3_examples/_sbr_examples/job_shop

Documentation for this example is found at

$SIMHOME/sim_examples/sim3_examples/_sbr_examples/job_shop/SIMSCRIPT_BusinessRulesInterpreter_Example.pdf