Scriptic Virtual Machine architecture

Scriptic comes with its own compiler and a virtual machine. The latter is a collection of Java classes that implement functions to which the compiler generates calls.
Essentially, the compiler creates for each script 3 items:

- A static template, with information about the script structure: the number of parameters, and the tree of operator types and operand types in the script expression. The template is initialized lazily using a getter method.
- a code method. This has a switch statement with pieces of code that the virtual machine may call. Such pieces of code are related to code fragments, script calls, if and while expressions, local data declarations etc.
- a method for parameter transfer. This has the same parameters as the script, plus an extra parameter. When called, this method packs its parameters into an array, and hands these to the scriptic virtual machine, together with a reference to the template.

More details on the compiler and the virtual machine follow.

Script compilation

Consider the following class and scripts:

```
public class T {
  public static scripts
      main(String args[]) = {System.out.print  ("Hello ")};
                            {System.out.println("world!")}
  public scripts
      A(int k) = int i: {i=0}; int j: B(i, j?, k?!,10!) + C + C1
      B(int i, int j, int k,int l) = D[i]<<(i) & D[1]>>(j?)
      D[10 i]<<>>(int j) >= {duration=10, priority=1:
                            System.out.println("D["+i+"]<<>>("+j
                                              +((j?)?"?":"")
                                              +( j! ?"!":"")+"):"
                                              +j!!)}
      C, C1 >= {}
      C, C2 >= {}
      C3, C >= {}
  }
```

Here main is the most simple script. Script main is compiled into code within class T that is essentially equivalent to the following:

```
public static void main(String args[])
{
    FromJava.mainscript().startScript(null, main_template(), new Object[] {
        args
    });
}

private static NodeTemplate main_template()
{
    if(main_template != null)
    {
        return main_template;
    } else
    {
        main_template = NodeTemplate.makeNodeTemplate("", "T", "main",
"([Ljava/lang/String;)", "T__main_code", 9, (short)0, (short)1, new
Object[][] {
            new Object[] {
                new int[] {
                    12, 0, 10, 7, 10, 26
                }, new int[] {
                    10, 7, 4
                }
            }, new Object[] {
```

```
                new int[] {
                    59, 0, 10, 29, 11, 59
                }, new int[] {
                    10, 59
                }
            }, new Object[] {
                new int[] {
                    123, 1, 10, 29, 10, 59
                }
            }, new Object[] {
                new int[] {
                    123, 1, 11, 29, 11, 59
                }
            }
        });
        return main_template;
    }
}

public static void T__main_code(Node node, int i)
{
    switch(i)
    {
    case 0: // '\0'
        System.out.print("Hello ");
        return;

    case 1: // '\001'
        System.out.println("world!");
        break;
    }
}
```

Notes:

1. The template function main_template () returns the template variable. It lazily ensures that the template variable is initialized properly. Template functions and variables are always static.
2. The code method T__main_code() is static because script main was static. This method is to be invoked dynamically from the Scriptic virtual machine; therefore the enclosing class must be public.
3. In case there were more scripts with the same name as main in class T, then most generated items in would have got a sequence number attached to the name main, such as main_000_template; only the generated script functions would all have the plain name main, because they will distinguish themselves with their unique list of parameter types.
4. In the template function there is a modifiers flag parameter. This accept standard Java VM values for the modifier bits, i.e., 0x0001=public, 0x0002=private, 0x0004=protected, 0x0008=static, 0x0010=final)

The Array in the Template Constructor Call

The last parameter to the NodeTemplate constructor is an array with 3 levels of nesting. Top level elements, such as the array
`{ { 12,0,3, 7,3,26}, {3, 7,4} }`, describe operators or operands in the parse tree of the script. Each such an array holds 1 up to 4 elements, which are in turn arrays. These have the following format and meaning:
element[0]: {token, codeBits, sl, sp, el, ep}

token - operator/operand type. In the example: 12, meaning Script declaration.

codeBits - flag field; a bit mask with the following possible bits:

Name	Value	Meaning
CodeFlag	0x0001	whether there is code
InitCodeFlag	0x0002	whether there is initialization code
NextCodeFlag	0x0004	whether there is 'next' code in 'for'
DurationFlag	0x0008	whether there is a duration

SecondTermFlag	0x0010	whether there is an 'else' for 'if' or an ':' for '?'
TrueFlag	0x0020	whether the expression is constantly true
FalseFlag	0x0040	whether the expression is constantly false
IterationFlag	0x0080	whether the template is an iteration
AnchorFlag	0x0100	whether the code has an anchor specifier
AsyncFlag	0x0200	whether the anchor is an asynchronous code invoker
SparseActivation Flag	0x0400	whether the activation of a communication must be sparse

sl – start line (in the example: 3)
sp – start position (in the example: 7)
el – end line (in the example: 3)
ep – end position (in the example: 91)

element [1], A: {l, p, len} or B: { l1,p1, l2,p2,..., ...}
l, p, len - line, position, length of central name (in the example: 3, 7, 4)
l1,p1, l2,p2, ... – lines, positions of n-ary operator symbols (lenght==1).
This determines the arity of the operator

element[2]: {"string"} in case of script calls (with name of called script) or in case of local data declarations (with name of local variable).

element[3]: {exclamParams, questParams, index, ...} for script calls
exclamParams - 64 bit mask for actual parameters with exclamation mark
questParams - 64 bit mask for actual parameters with question mark
index, ... – for each adapting parameter (having both question mark and exclamation mark) its index in the formal parameter list of the enclosing script

The following table defines what token values are possible, and what further elements are applicable:

Value	**Meaning**	**1A**	**1B**	**2**	**3**
12	Script declaration	X			
13	Communication declaration	X			
14	Channel declaration	X			
15	Send channel declaration	X			
16	Receive channel declaration	X			
'{'	Code fragment				
'.'	Event handling code fragment				
23	Looping event handling code fragment				
19	Tiny code fragment				
20	Unsure code fragment				
22	Threaded code fragment				
'<'	Activation code	X			
'>'	Deactivation code	X			
'?'	Conditional script expression		X		
'i'	If script expression		X		
'w'	While script expression	X			
'f'	For script expression	X			
's'	Switch script expression	X			
't'	Case tag script expression	X			
'c'	Script call expression	X		X	X
17	Channel send expression	X		X	X
18	Channel receive expression	X		X	X
'd'	Script local data declaration	X		X	
'p'	Private script data declaration	X		X	
24	Deadlock process				
25	Empty process				
26	Break expression				
27	Ellipsis operand				
29	Launched expression				

31	Ellipsis operator		X		
32	Channel request				
33	Communication request				
'/'	break operator		X		
'|'	parallel or operator		X		
'&'	parallel and operator		X		
'&'+1	parallel and2 operator		X		
'"'	parallel or2 operator		X		
'+'	or operator		X		
';'	sequence operator		X		
'%'	not-sequence operator		X		
'-'	reactive not operator				
'~'	not operator				

For script A(int k) = int i: {i=0}; int j: B(i, j?, k?!,10!) + C + C1 the generated items are:

```
private static scriptic.vm.NodeTemplate A_template;
public void A(Node node, int i)
{
    ((CallerNodeInterface)node).startScript(this, A_template(), new Object[]
{
        new Integer(i)
    });
}

private static NodeTemplate A_template()
{
    if(A_template != null)
    {
        return A_template;
    } else
    {
        A_template = NodeTemplate.makeNodeTemplate("", "T", "A",
"(Lscriptic/vm/Node;I)", "T__A_code", 1, (short)0, (short)1, new Object[][]
{
            new Object[] {
                new int[] {
                    12, 0, 13, 7, 13, 15
                }, new int[] {
                    13, 7, 1
                }
            }, new Object[] {
                new int[] {
                    43, 0, 13, 18, 13, 65
                }, new int[] {
                    13, 57, 13, 61
                }
            }, new Object[] {
                new int[] {
                    100, 1, 13, 22, 13, 23
                }, new int[] {
                    13, 22, 1
                }, new Object[] {
                    "i"
                }
            }, new Object[] {
                new int[] {
                    59, 0, 13, 18, 13, 56
                }, new int[] {
                    13, 30
                }
            }, new Object[] {
                new int[] {
                    123, 1, 13, 25, 13, 30
                }
```

```
            }, new Object[] {
                new int[] {
                    100, 1, 13, 36, 13, 37
                }, new int[] {
                    13, 36, 1
                }, new Object[] {
                    "j"
                }
            }, new Object[] {
                new int[] {
                    99, 1, 13, 39, 13, 56
                }, new int[] {
                    13, 39, 1
                }, new Object[] {
                    "B"
                }, new long[] {
                    6L, 12L, 0
                }
            }, new Object[] {
                new int[] {
                    99, 1, 13, 59, 13, 60
                }, new int[] {
                    13, 59, 1
                }, new Object[] {
                    "C"
                }, new long[2]
            }, new Object[] {
                new int[] {
                    99, 1, 13, 63, 13, 65
                }, new int[] {
                    13, 63, 2
                }, new Object[] {
                    "C1"
                }, new long[2]
            }
        });
        return A_template;
    }
}

public void T__A_code(Node node, int i)
{
    switch(i)
    {
    case 0: // '\0'
        node.setLocalData(0, new IntHolder());
        return;

    case 1: // '\001'
        ((IntHolder)node.localData[0]).value = 0;
        return;

    case 2: // '\002'
        node.setLocalData(1, new IntHolder());
        return;

    case 3: // '\003'
        B(node, ((IntHolder)node.localData[0]).value,
((IntHolder)node.localData[1]).value,
((IntHolder)node.paramData()[0]).value, 10);
        return;

    case 4: // '\004'
        ((IntHolder)node.localData[1]).value =
((IntHolder)node.calleeParams()[1]).value;
        return;

    case 5: // '\005'
```

```
        ((IntHolder)node.paramData()[0]).value =
((IntHolder)node.calleeParams()[2]).value;
        return;

    case 6: // '\006'
        C(node);
        return;

    case 7: // '\007'
        C1(node);
        break;
    }
}
```

For script `D[10 i]<<>>(int j) = {duration=10, priority=1: System.out.println("D["+i+"]<<>>("+j +((j?)?"?":"") +( j! ?"!":"")+"):" +j!!)}`

the generated items are:

```
private static scriptic.vm.NodeTemplate D_template;
private static scriptic.vm.NodeTemplate D_chan_template;
private static NodeTemplate D_chan_template()
{
    if(D_chan_template != null)
    {
        return D_chan_template;
    } else
    {
        D_chan_template = NodeTemplate.makeNodeTemplate("", "T", "D",
"(Lscriptic/vm/Node;II)", "T__D_chan_code", 1, (short)1, (short)1, new
Object[][] {
            new Object[] {
                new int[] {
                    14, 1024, 15, 7, 15, 26
                }, new int[] {
                    15, 7, 1
                }
            }, new Object[] {
                new int[] {
                    123, 11, 15, 30, 19, 53
                }
            }
        });
        D_chan_template.setRelatedTemplates(new NodeTemplate[] {
            D_template()
        });
        return D_chan_template;
    }
}

private static NodeTemplate D_template()
{
    if(D_template != null)
    {
        return D_template;
    } else
    {
        D_template = NodeTemplate.makeNodeTemplate("", "T", "D",
"(Lscriptic/vm/Node;II)", null, 1, (short)1, (short)1, new Object[][] {
            new Object[] {
                new int[] {
                    6, 1024, 15, 7, 15, 26
                }, new int[] {
                    15, 7, 1
                }
```

```
            }
        });
        D_template.setRelatedTemplates(new NodeTemplate[] {
            D_chan_template()
        });
        return D_template;
    }
}

public void T__D_chan_code(Node node, int i)
{
    switch(i)
    {
    case 0: // '\0'
        node.setDuration(10D);
        node.priority = 1;
        return;

    case 1: // '\001'
        System.out.println("D[" + ((IntHolder)node.paramData(0)[0]).value +
"]<<>>(" + ((IntHolder)node.paramData(0)[1]).value + (node.isOut(1) ? "?" :
"") + (node.isForced(1) ? "!" : "") + "):" + (node.isForced(1) ?
((IntHolder)node.paramData(0)[1]).hasEqualValue((Integer)node.oldParamData(1
)[1]) : true));
        break;
    }
}
```

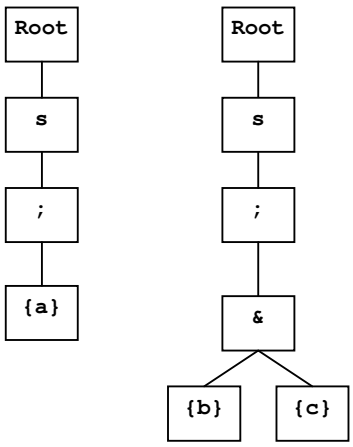
Code generated for C, C1, C2 and C3 is likewise.

Virtual Machine

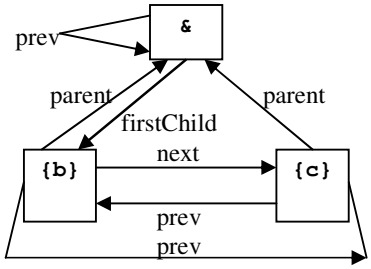
The Scriptic Virtual Machine maintains data structures that ensure that calls are made to the generated code functions at appropriate times. The main structure in the virtual machine is a run-time hierarchic graph of Node instances. This dynamic graph is constructed using information from static template trees. These are generated once for each script upon its first call, using the call to the makeTemplate method in the compiled code. Executing a Scriptic program means that in passes this run-time graph is expanded (activation) and shrinked (deactivation). At appropriate times, fragments of the compiled code are executed in the switch statements of the code functions in the classes generated for the scripts. The virtual machine has been optimized for speed, not for memory usage or simplicity.

The Run-time Graph

The top of the run-time graph is always the so-called RootNode, and at the second level there is a node corresponding to a script being called from Java, or the main script called from the Java virtual machine. Example: when executing the script `s = {a};({b}&{c})` the graph looks subseqently like:



This graph is maintained using using references in Node instances named parent, prev, next and firstChild. E.g., to pick out the portion {b}&{c}, the following references apply:



The prev-next references implement a semi-closed double linked list, which eases dynamic insertion and removal of nodes.

There are more references within the run-time graph:
refinementAncestor: points to the nearest ancestor node that is a script or communication refinement. For nodes s and below, this reference points to s.
operatorAncestor: points to the nearest ancestor node that is an n-ary operator (|| | && | / % ; +)
reacAncestor: points to the nearest ancestor that is one of the operators. - or %
subRootNode: points to the ancestor node just under the RootNode.

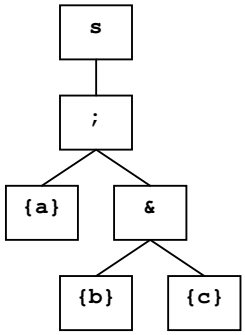
n-ary operator nodes also have a flag telling whether the node is actually an iteration..

A script call results into two nodes: a parent node for the call with the actual parameter list, and a child node for the script with the formal parameter list being called.

Each node in the run-time graph has a reference to the frame variable that was created for the script in the code generated by the compiler. The main purpose of the frame is to provide the virtual machine a hook at which compiled code fragments and script calls may be invoked in the switch statement of the code function in the frame class. The virtual machine my calls something of the kind:
`theNode.frame.code(theNode, theIndex)`

Template trees

The run-time graph is modelled after information from static template trees. At the first call of a script, static information about its structure is packed into a nested array and handed to the virtual machine in a constructor call to class NodeTemplate. For script s the template looks like:

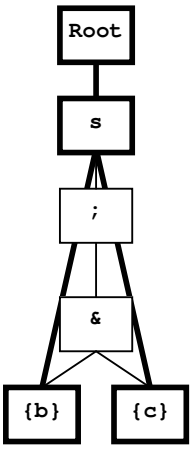


In the template tree each node has a reference to its parent. The set of child nodes of a parent node is managed by an array at the parent. Each child node knows at what index in this array it is referenced to. In such a static structure an array is faster than a double linked list since no dynamic insertion and

deletion of nodes is needed. Each node has a type code, denoting the corresponding script operator (|| | && & +), operand or refinement (script or communication declaration). Nodes associated with pieces of compiled Java code (such as code fragments and script calls) know the appropriate index or indexes in the switch statement of the code function in the class generated for the corresponding script.

Parallel operands tree

As a kind of summary of the run-time graph, there exists a parallel operands tree. The nodes of this tree correspond to operands in the run-time graph of parallel operators: || | && & and /, and to the root node, the childs of the root node and the childs of RSC. E.g. for the second figure of the run-time graph, the overlaying the parallel operands tree in bold face results in:



Like the main run-time graph, the parallel operands tree is maintained through references named parent, firstChild, prev and next. Each node has a reference named node to corresponding nodes in the run-time graph. There is an inverse reference named parOpAncestor from the run-time graph to the tree. In the example, the parOpAncestor references for the semicolon and ampersand nodes point to the s node in the tree.

Requests

A *request* is either a code fragment node or a call to a communicating script, in the run-time tree. A reference to it is stored in a list using references prevInParOp and nextInParOp, at a node in the parallel operands tree pointed to by its parOpAncestor reference. In the example, requests {b} and {c} are stored in lists at the corresponding position in the parallel operands tree. When the ampersand had been a plus, they would have been stored at s.
The purpose of the parallel operands tree is to enable a fast algorithm for determining whether a given pair of requests is exclusive or not. Two requests are exclusive if they have been activated in different branches of a node that is either ; % +. It follows that they are exclusive when their parOpAncestor nodes are equal or when one of the parOpAncestor nodes is in the subtree of the parallel operand tree that pends under the other parOpAncestor node.
References to requests are also stored in so-called request lists, with reference fields prevReq and nextReq. For the purpose, see later.

Activation

```
((CallerNodeInterface)node).startScript(
      this, A_template(), new Object[] {new Integer(i)});
```

A new Node corresponding to a given template is usually created by calling the function activateFrom(the parent node) at the template. This creates a node of the appropriate class, and puts it in the run-time graph. In case of some operator types (for, while, if, switch, break, {:}, .., -) no new node is created; the activation then has a special semantics.
To cope with the semantics of the activation, the ActivateFrom function returns a bitSet with such bits as:

BitField	Description

CodeBit	has any (non-tiny) code fragment been activated
SuccessBit	does this activation result in a success for the parent, e.g., with "-" as an operand, or with {:} and ".." in an and-like context
SomeBit	has ".." been activated
ExitBit	has "break" been activated

The activator passes this bitset onwards, or, in case of an n-ary operator, takes appropriate action.
After such a node has been created, the function activate is called on the node.
In many cases this calls in turn again activateFrom at a child of the node template.
In case of a request node, activate() puts it into the parallel operands tree, and in an appropriate request list.
In case of a script call node, activate() executes the associate code in the calling script class, which calls in turn the called script as a function, with the calling node as extra first parameter. E.g., suppose script A calls script B. Activating this call is done using the following call stack:

Function	**Description**
template.ActivateFrom(parent)	The node template with type 'c' (call), corresponding to the call to B, is activated, creates a ScriptCallNode and puts it under the parent in the run-time graph
node.activate	node is the just created ScriptCallNode
node.doCode	determines the appropriate index in the switch statement of the code function in the script frame
node.frame.code(node,index)	node.frame is of class A
B(node)	in the compiled code for script A (code function)
node.startScript(B_frame, params)	in the compiled code for script B (script function)
B_frame.template.activateFrom(node, frame,params)	depending on the template's type, creates a ScriptDeclarationNode or something similar
childNode.activate	etc. (childNode is the newly created node)

try-out

An activated code fragment node is put in a request list, sorted on priority and duration. Such code fragments are tried out at a stage when no more activations are imminent. Trying out means that the generated code function in the script class gets executed with the appropriate index. If the code fragment node is an unsure one, the attribute success is inspected afterwards; if it is false, the code fragment has failed and it stays in the request list. Otherwise, the code fragment has success. It then exists its request list and enters a success list, for later processing. Moreover all request nodes that are exclusive with the succeeded one are deactivated. These request nodes are identified using the parallel operands tree.
Trying out is more complicated in case the code fragment is the first one to be executed inside a communication, see later.

success

After trying out nodes the success list is handled. Of the request nodes in that list with the highest priority the lowest duration is chosen. This duration is the simulation time that may pass by. The exact amount of simulation time that passes by may be decreased by a call to the function scriptic.vm.FromJava.tick(AttributesInterface), e.g. to establish a relation between simulation time and real time.
Then the simulation time that has passed by is subtracted from the duration of the nodes in the success list with the highest priority. Then the nodes with the highest priority and zero duration are taken from the success list. First the function succeed() is called on them, which transfers a success message in the run-time graph upwards. Such success messages may also be transferred upon activation, such as with the empty process (-).
It depends on the parent in the graph what happens with the success message. E.g., if it is a sequence operator node (;), and if the success comes from a child that is not the rightmost one in the static template tree, then a child is activated according to the next template. E.g., activating -;x activates the empty process first. This has a success which is reported through the successbit (rather than via the success list). Then, during the same activation cycle, x is activated.
In {a};x a successful try-out of {a} also results in activating x, but only at a later stage, in the function Node.succeed. A success by x (the right-most operand) will be propagated upwards
by the sequence node, to its parent.

Finally these succeeded request nodes are deactivated.

deactivation

Nodes may be deactivated for different reasons.
First consider leaf nodes: a code fragment having been tried out successfully, gets deactivated thereafter. This is a so-called active deactivation. Alternatively, a code fragment or a communication request may also be excluded, causing an inactive deactivation.
Deactivation propagates to parent nodes as long as they have no more active childs.
Active deactivations may result in activations, as with the % operators, or in successes, as with the negation operators.

the main loop

The way so start a script from Java is to call it as if it were a function with extra first parameter scriptic.vm.FromJava.mainscript(). If the script to be called is a public static script main(String args[]), which should be the entry point in the Java program, then this call to scriptic.vm.FromJava.mainscript() is generated by the compiler in the code of the script function for main.
scriptic.vm.FromJava.mainscript() returns a reference to the RootNode. The call to startScript activates the script nodes, and then calls scriptic.vm.FromJava.mainLoop(). This repeatedly calls the boolean function hasActivity at the RootNode, until it returns false.

communication

Frames are a structure between activated scripts and templates, to determine whether communication can occur. There are frames for the communicating partners, and for the communication bodies.
Partner frames are owner of request lists with communication requests (of the same associated script).
E.g., activating main() in

```
scripts
  A,B  = ....
  main = A,A,A, ...
```

would initially enter 3 CommRequests (for A) into A's frame.
A partner frame with empty request list is called inactive; otherwise it is called active. Communication frames for which all associated partner frames are active, are also called active; otherwise they are inactive. An active communication frame has an associated trial tree pending under the RootNode: a CommNode with child nodes activated from the template. This tree is not yet bound to specific partner requests. That changes after successfully trying out a code fragment for the first time in this tentative tree.
When a partner frame changes from active to inactive or back, this may have impact on the state of the associated communication frames. At such moments trial trees may be activated or deactivated.
Tentative communication always starts with a 'trial tree', which is not yet bound to communicating partners. Initially its activated code fragments are not bound either. Such unbound code fragments are tried out by binding them tentatively to sets of partners, yielding a proper parameter context. If the code fragment is unsure, and the trying out fails, then the code fragment is not deactivated; it may be retried again.
An active deactivation of a communication script results in active deactivation of the involved partner communication requests, so propagation continues.