Implementation of a Design Virtual Machine

TR 29.3022

Stephen L. Burbeck

Stephen G. Graham

Emerging Technologies
IBM Network Computing Software
Research Triangle Park, North Carolina 27709

The authors have used their best efforts in the preparation of this report. International Business Machines Corporation and the authors make no representation or warranties with respect to the accuracy or completeness of the contents of this report.

The following terms, denoted by ® or ™, used in this report are registered trademarks or trademarks of International Business Machines Corporation in the United States and other countries.

IBM
VisualAge
ThinkPad

The following terms, denoted by ® or ™, used in this report are registered trademarks or trademarks of the listed holder. All other brand names and product names used in this report are trademarks, registered trademarks, or trade names of their respective holders.

Java	Sun Microsystems, Inc.

ABSTRACT

This report describes the design and implementation of a design virtual machine that “executes” fine grain design descriptions of Smalltalk systems. The design statements, in the form of signatures and qualifiers, characterize method level collaborations between Smalltalk objects. The code specifies the details of those collaborations. Code is compiled into the design analogs of bytecodes that are executed by the design virtual machine. The execution of the design either validates the collective design declarations involved in the execution of a method, or finds mismatches between the design and the code. This process is related to but far more challenging than type checking in strongly typed languages. The static analysis of a typical method takes less than 100 milliseconds on a modern PC and therefore can be done in real-time whenever any method is browsed or changed.

It is assumed that the reader is familiar with programming in Smalltalk and has access to the separate IBM Technical Reports on Signatures and on Static analysis of Smalltalk Signatures (see Bibliography).

ITIRC KEYWORDS

- Design virtual machine
- Executable design
- Smalltalk
- Data types
- Signature

CONTENTS

INTRODUCTION

In a separate IBM Technical Report [BG98a], we argue that code and design are fundamentally different descriptions of an object oriented (OO) system. Both are expressions of human intent that must be maintained separately. To use a theatrical metaphor, code specifies the dialog and actions that take place in a scene whereas design characterizes the actors' roles and responsibilities.

Tools are needed to keep code and design synchronized as changes are made to the system. The collaboration web in OO systems, especially Smalltalk, is so complex that changes made to one method can violate the design in many others. Without tools to help pinpoint such problems, the code in a system quickly diverges from its explicit design description.

We assert that the best approach to keeping code synchronized with design is method-by-method static analysis. Just as code in a Smalltalk system is distributed among a large number of small methods, so is design. We distribute behavior into many small methods rather than fewer large methods so that each method does one thing and one thing well [JF88, Bur96]. Each of these "things" is ideally the implementation of one responsibility of an object, so its design is an atomic portion of the design of the system. We need, therefore, to analyze the portion of the design relevant to each method in the context of the code in that method.

Prior approaches to static analysis have been based upon the parse tree (see [BG98a, BI82]). We use a different approach – a design virtual machine – largely because of the difficulties of dealing with literal blocks. Here we summarize the reasons for favoring a design virtual machine and the basic theory of the approach before presenting the details of our approach.

The challenge of literal blocks

Static code analysis of a Smalltalk method is much like the process a programmer takes when reading the code. As we read through code, we mentally parse it, keep track of which objects are assigned to variables, and note whether objects returned from messages sent are then assigned to variables or are sent subsequent messages. We view the objects involved in the method more as generic instances of the appropriate class rather than specific instances with specific values of instance variables.

When literal blocks are involved, the task becomes more difficult both for reading the code and for static analysis. Literal blocks are essentially unnamed methods that are defined in-line and invoked by a variety of mechanisms that can be quite subtle. When we read code that contains blocks, we have to mentally step back and determine when the block will be evaluated, under what conditions, and with what arguments. Then we envision the block being evaluated, perhaps iteratively. We also must be sure we understand the fate of the result of each evaluation of the block (especially with messages such as #collect:, or #inject:into:). In some cases, e.g., SortedCollection >> #sortBlock:, the block is not

evaluated until some undetermined later time. Nonetheless, when we read the code in which the #sortBlock: message is sent, we mentally "execute" the sort block to see how the sorting will be done.

Literal blocks also complicate matters for static analysis. Blocks may be assigned to variables that later are sent 'value' messages. These blocks are not invoked where they appear in the parse tree. Also, a literal block may be invoked with different arguments within the same method. The following example illustrates some of these issues. It has been contrived specifically to challenge a static analyzer. It also is quite a challenge for human analysis to determine whether the method can execute and, if so, whether its signature is correct. If the reader wishes to take that challenge, note that the *conditionalBlockDepth* method returns a nonnegative integer.

```
twistedBlockValues
      "<^hOf Integer>"
      "Example of sending value to literal blocks.
       Note: evaluation order is very different from appearance order and
                 blocks are entered multiple times."

      | firstBlock secondBlock thirdBlock twistBlock |
      twistBlock := [:blkArg1 :blkArg2 | blkArg1 value: blkArg2 value].
      firstBlock := [:myArg | myArg * secondBlock value].
      secondBlock := [self conditionalBlockDepth].
      thirdBlock := [:myArg | myArg printString].
      ^(thirdBlock value: secondBlock value) size > 3
            ifTrue:
                  [twistBlock value: thirdBlock value: secondBlock]
            ifFalse:
                  [twistBlock value: firstBlock value: secondBlock]
```

In this contrived example, the static analysis must not and cannot analyze the blocks when they first appear. At the point they appear in the parse tree, nothing is known about what kinds of objects will occupy the blockArgs. Moreover, the analyzer must pass through the blocks each time they are invoked because the types of the blockArgs may (and do) differ when invoked from different places. No one-pass traversal of the parse tree can analyze this code. The design virtual machine approach we describe in this report can handle such twisted code (it analyzes the twistedBlockValues method in 109 milliseconds on a 133 MHz IBM ThinkPad™ and, by the way, its signature should be `<^(hOf Integer | hOf String)>`). The design virtual machine can also analyze the wide variety of blocks involved in iteration (e.g., #collect: and #inject:into:) and some kinds of recursively invoked blocks.

EXECUTABLE DESIGN

The central idea of this report is that static analysis can be approached in a manner analogous to the way the virtual machine executes compiled Smalltalk code. To explain this approach we must first examine the way Smalltalk virtual machines execute code.

Three categories of activities occur during execution: memory management (i.e., the creation and destruction of objects), handling of external events, and interpretation of bytecodes that have been previously compiled from Smalltalk methods. For the purpose of static analysis, we can ignore memory management and external events because the visible behavior of Smalltalk code occurs under the explicit direction of bytecodes.

The compiler converts Smalltalk source code to bytecodes. These bytecodes are interpreted by a virtual machine [GR83] [Ing83]. The virtual machine is a stack machine. Bytecodes direct the virtual machine to push objects from variables (e.g., method args, instance variables, etc.) onto a stack, pop objects from the stack to store them into variables, and send messages. Message sends pop their arguments and the receiver from the stack and push the result onto the stack. Each invocation of a method or a block creates a MethodContext or BlockContext object that maintains an instruction pointer into its bytecodes and provides private state. A second virtual machine stack manages the active contexts during execution. This second stack is the one made visible in the Smalltalk debugger.

Recall that when we read code to understand its effect, we mentally replace the actual objects with generic stand-ins. In our design virtual machine, qualifiers stand in for the objects so described. Signatures similarly stand in for methods invoked as a result of a message send. That is, objects and message sends are well described by qualifiers and signatures respectively (see [BG98a, BG98b]). The question is how to represent the behavior of the code itself. Our approach, rather than representing the code as a parse tree, is to represent it as analogs of bytecodes (hereafter called *execution steps*) that are created from the parse tree in a manner very similar to the generation of bytecodes.

In the design virtual machine approach to static analysis, a specialized virtual machine *executes the design, as specified by signatures and qualifiers, in the context of the code as specified by the execution steps*. That is, where objects are pushed and popped to and from variables at runtime, qualifiers are pushed and popped to and from attributes by the design virtual machine.

A runtime virtual machine executes message sends by pushing the receiver and args then looking up the method to invoke, beginning in the class of the receiver and working up the inheritance chain (note: if the receiver is "`super`" the lookup begins in the superclass of the receiver). When the invoked method returns, its return value is left on the stack. The runtime evaluation of blockContexts is similar to a message send because blocks are essentially unnamed methods. However, no method look-up is needed

A design virtual machine executes a message send in an analogous manner. Under the direction of execution steps, it begins execution of a method by pushing a self qualifier

followed by arg qualifiers. Method look-up differs from the runtime technique in one important way: if the receiving qualifier is an alternatives qualifier, we must analyze the signatures of all of the potentially receivers. Note, however, that we do not need to recursively analyze the method(s) that would be invoked. The design virtual machine need only check that the argument qualifiers on the stack are valid according to the qualifiers in the receiving signature(s). After this process, the return qualifier, determined by the signature(s) of the method(s) that would be invoked, is left on the stack. The evaluation of blocks by the design virtual machine proceeds in a manner similar to a message send.

The following table shows the elements involved in execution of Smalltalk in a typical virtual machine and the corresponding elements in the design virtual machine.

Execution Model	Design Model
Objects	Qualifiers
Message sends	Signatures
Variables	Attributes
Virtual Machine	ExecutionModel
Block/Method Contexts	ExecutionContexts
Bytecodes	ExecutionSteps

Table 1. Correspondence between elements of the execution VM and the design VM

Qualifiers

Qualifiers are the design equivalents of objects. A qualifier specifies, via the qualifier syntax given in [BG98b], what kind of object can be encountered at runtime. The simpler ones state that the object will be an instance of some specific class (denoted by *instanceOf aClass*, abbreviated as *iOf aClass*) or an instance of some class or its subclasses (indicated as *hierarchyOf aClass* abbreviated as *hOf aClass*). Qualifiers can also specify the three "special" objects: *true*, *false*, and *nil*. Other qualifiers allow one to describe objects whose qualification depends upon the context in which they are found, e.g., *self*, *super*, or *arg1*. More complex qualifiers can specify such things as:

- the object is the class itself (i.e., *iOf class aClass*),
- the object satisfies one of a list of alternative qualifiers (e.g., *(hOf Integer | nil)*),
- the object is a copy of the second argument to the method (denoted *arg2 copy*),
- the object is a collection that holds objects that satisfy a qualifier (e.g., *iOf OrderedCollection {of: (hOf CustomerRecord | hOf SalesInvoice}*),

- the object is a one argument block where the block argument must satisfy a qualifier and the block returns an object that satisfies another qualifier, (e.g., *[:blockArg1 hOf CustomerRecord, ^(true | false)]*).

Qualifiers such as *self* or *arg1* have two special properties not shared by the other qualifiers. First, they refer to specific objects, i.e., instances, not just to their class (hence, we call them *instance qualifiers*). That is, when a method's signature says that it returns arg1 (e.g., *Collection>>#add:*), the signature means that the method returns the *same* instance that it receives as its first argument, not just that it returns an instance of the same class. Thus the static analysis must be able to determine whether the same instance was returned and not be fooled by the return of an object of the same class as the first arg, or even by a copy of the first arg. Second, these qualifiers are *unresolved*, i.e., we cannot know the properties of the object until we can resolve those properties in the context of the analysis.

The qualifiers that describe constituent parts of other qualifiers are called *aspect qualifiers*, i.e., they qualify some aspect of the object described by another qualifier. In the example mentioned above, *iOf OrderedCollection {of: hOf CustomerRecord | hOf SalesInvoice }*, the {of: …} modifier is the aspect qualifier that describes the elements of the collection. When the aspect is an alternatives qualifier as in this case, the alternatives need not have any inheritance relationship to each other. Some kinds of object such as Dictionary or Stream have more than one aspect. A Dictionary needs a *key:* aspect and a *value:* aspect. Streams need two aspects for a more subtle reason; Stream protocol includes methods that deal with single elements of the collection managed by the stream (e.g., *#next*, or *#nextPut:*) as well as methods which refer to the whole collection (e.g., *#contents*). Thus Stream requires the *of:* aspect to refer to the individual elements as well as the *on:* aspect to refer to the whole collection. Conveniently, however, most uses of stream can rely on the defaults for these aspects, which describe a *String* of *Characters*.

Much of the operation of the design virtual machine only requires that qualifiers be manipulated on a stack, e.g., pushed or popped, and shuttled between the stack and the various attributes involved in a method. But qualifiers play a more active role in modeling the sending of messages. The qualifier is responsible for determining which methods might be invoked when a given message selector is sent to the object represented by the qualifier. If the qualifier models instances of a single class, the method lookup starts at that class. If the qualifier specifies multiple alternatives, the lookup is done separately for each alternative. If the qualifier depends on its context, e.g., self, then that context is used to resolve the qualifier to an instance of some class before the lookup proceeds.

It should be noted that the problems posed for static analysis by instance qualifiers, aspect qualifiers, and block qualifiers, taken singly or in combination, go far beyond the issues involved in type checking C++ or Java™ code.

Signatures

Signatures model the external aspects of a method: the kinds of objects that are sent to a method (i.e., its args) and returned from a method. A qualifier specifies the "kind" of each of these objects. The signature is a list of qualifiers delimited by angle brackets. There is one qualifier for each argument to the method plus on for the return from the method. During

analysis of a method, signatures play two roles. The signature of the method under analysis provides the argument qualifiers for the method we are analyzing, which tell us what kind of objects enter as arguments. Signatures from the methods invoked by code in the method provide the information necessary to analyze each message sent and determine the objects returned from those message sends. These signatures allow us to check that the qualifiers of the arguments satisfy the corresponding qualifiers of the methods to be invoked, and allow us to determine the qualifier (which may be an alternatives qualifier) of the result of the message send.

It is common for people, on first encountering the notion of static analysis, to believe that the analysis must recursively analyze all the message sends invoked during the execution of the method, i.e., also analyze all the methods that would be executed at runtime as a result of executing the method under analysis. This is not so. We assume that the signatures of the methods called by the target of analysis are correct. We determine the correctness of those signatures in the context of their methods by separate analyses.

Attributes

Attributes model variables: method and block arguments, method and block temps, instVars, classVars, poolVars, and globals. In general, attributes contain two qualifiers, a *declared qualifier* that specifies design intent and a *deduced qualifier* that keeps track of which objects are assigned to the variable during static analysis. Temporary variables lack a declared qualifier because there is no mechanism by which the developer can declare design intent for those variables. And method arguments lack a deduced qualifier because objects may not be assigned to them during execution.

Whenever a qualifier is assigned to an attribute (including the attribute for the method return), the analysis determines whether the qualifier assigned to the attribute is legal according to the declared qualifier. That is, are the kinds of objects implied by the deduced qualifier a subset of those allowed by the declared qualifier? If not, an error is signaled to the design virtual machine.

The ExecutionModel

The basic role of the ExecutionModel is to stand in for the runtime behavior of the virtual machines in its execution of a single method. But the ExecutionModel must do additional work imposed by the fact that the stand-ins for objects and messages, i.e., qualifiers and signatures, are somewhat more complex than are objects and messages at runtime. The ExecutionModel provides the following behavior:

- It manages all the attributes involved in the method. These may include an attribute for any variable that is in the scope of the method (e.g., arguments, temp variables, Pool variables, Class variables, or globals) as well as an attribute for the method's return value.
- It resolves unresolved qualifiers e.g., self, super, and argN, in the context of the execution. These qualifiers may occur in qualifiers involved in a message send as well as in the signature of the method being analyzed.
- It manages the two stacks. The qualifier stack holds the qualifiers that are pushed during execution. The context stack holds the execution contexts that are active, i.e., one for the method itself, and one for each block that has been entered and not yet exited.
- It executes the executionSteps that direct the actual execution. Each execution context maintains a currentStep pointer. The ExecutionModel iteratively executes the current step of the executionContext that is on the top of the contextStack.
- It records notifications generated during execution. These are of three types: informational, warning, or error.

The only informational notification is that a receiver has been made more concrete. Warnings are generated if a method improperly calls another method that is marked 'private', if a method contains a block that has not been evaluated, or if an argument qualifier for a message send contains some alternatives that the receiving method cannot handle along with some that it can. Thus, analysis is optimistic. It assumes that if some alternatives are qualified, the method may work. The warning, however, lets the user decide whether these improper alternatives will actually be sent as arguments. Errors are signaled whenever a disqualification occurs, i.e., when the deduced qualifier of an attribute does not meet the qualification of the declared qualifier, or when the deduced qualifier for the receiver of a message does not understand the message. Here again, if the receiver qualifier is an alternatives qualifier and some of the alternatives qualify, only a warning is generated.

ExecutionContexts

There is one executionContext for the method and one for each literal block in the method. The context for the method is placed on the contextStack at the start of analysis. The

contexts for any literal blocks are created from the parse tree and attached to the *PushLiteralBlock* executionStep. Each executionContext contains the executionSteps for the code in the method or block, and holds attributes for any arguments or temp variables belonging to the method or block. It keeps track of which executionSteps have been executed and will convey upon request the next step to be executed. When a PushLiteralBlock step is executed, it pushes its executionContext onto the context stack. The next step to be executed, then, is the first step in this context, which is now on top of the context stack.

ExecutionSteps

Execution steps control most behavior specific to the code. There are twenty-six subclasses of the abstract ExecutionStep. Most deal with the qualifier stack, e.g., pushing, popping, fetching, storing, or returning qualifiers. Some are housekeeping steps such as those that cause the VM to enter and exit executionContexts. The most complex deal with message sends. The behavior of the different varieties will be discussed in more detail in a later section.

The behaviors that all executionSteps share are:

- Conveying the positions in the source string of the first and last characters of the code that generates the step. These positions are used to highlight code in the user interface if the user is watching the execution.
- Holding the syntactic element that is key to the operation of the step. For example, if the step fetches or stores to a variable, the key element is the variable name token. In a message send, it is the message pattern.
- Storing the result of the execution of the step.
- Marking the step as a breakpoint, i.e., a point where the execution stops (this is used for debugging the design virtual machine).

DESIGN AND IMPLEMENTATION OF THE VIRTUAL MACHINE

Outline of operation

An instance of SEMExecutionModel manages the static analysis of a method The SEM prefix stands for Semantic Execution Model and is used for all the classes in this application although we will usually refer to these classes without the prefix where no ambiguity can arise. An instance of NAFMethodArtifact, an object that wrappers the source code of a given method, creates the ExecutionModel based on the parse tree obtained from the system compiler (in this case, the compiler in VisualAge™). It requests the parse tree to generate executionSteps and add them to the executionModel's homeContext. These steps are then executed one at a time until the ExitMethodContext step is executed. Let's examine this process in somewhat more detail.

Creating the ExecutionModel

A new execution model is created by the following code sent to the instance of NAFMethodArtifact that represents the method.

```
NAFMethodArtifact>>
asExecutionModel
   "<^hOf SEMExecutionModel>"
   "Return a SEMExecutionModel on my method."

   | newModel homeContext |
   newModel := SEMExecutionModel new.
   newModel methodArtifact: self.
   homeContext := SEMExecutionContext forModel: newModel.
   newModel homeContext: homeContext.
   newModel externalStack: newModel prototypeStack.
   newModel initializeAttributes.
   self parseTree addStepsTo: homeContext.
   ^newModel
```

Notes:

- The externalStack contains the qualifiers for the args and receiver of the method. The prototypeStack method creates and returns an initial externalStack, i.e., an OrderedCollection with qualifiers for self and the args, self pushed first, then arg1, arg2...
- 'initializeAttributes' sets up attributes for the return value and any args, and resolves any instance qualifiers.

- Then the homeContext (an ExecutionContext) is created and the execution steps implied by the code are added to it by sending '*addStepsTo: homeContext*' to the root of the parse tree. The root (an instance of EsMethod in VisualAge) initiates a depth first traversal of the parse tree in which each node creates and adds its steps to the homeContext via its implementation of the #addStepsTo: message. Any literal block nodes in the parse tree create their own BlockContext and add their steps to that context, not to the home context (see later section on generation of steps from the parse tree for more detail).

Stepping through the static analysis

The executionModel steps to its end by executing its executionSteps one at a time until it encounters the ExitMethodContext step or encounters a mismatch between the code and the design (called a disqualification). Each step is executed by invoking the "nextStep" method. The two methods are as follows:

```
stepToEnd
   "<^(true | false)>"
   "Repeatedly step until end or disqualification.  Return true if
    reaches end OK, false if disqualification."

   | nextStep |
   nextStep := self nextStep.
   [nextStep isMethodExit or: [self isDisqualified]]
      whileFalse:
         [nextStep := self nextStep].
   ^self isDisqualified not


nextStep
   "<^hierarchyOf SEMExecutionStep>"
   "Execute and return result of next step.  If it has a breakpoint
    set, halt for debugging."

   | nextStep |
   nextStep := self activeContext nextStep.
   nextStep isVisible
      ifTrue: [self visibleStepCount: self visibleStepCount + 1].
   nextStep breakpoint
      ifTrue: [self halt].
   ^nextStep executeIn: self
```

Notes:

- The active context is the one on the top of the executionModel's contextStack. The nextStep method fetches the nextStep from the active context. The last step in a block context is an ExitBlockContext step which pops the context stack. The next outer context therefore resumes execution. A method context ending signals the end of the SEM analysis.
- Some steps are not "visible". Examples of non-visible steps are: entering and leaving contexts, or duplicating the top of the stack in preparation for a cascaded message send. For UI purposes, we need to keep track of the number of visible steps executed.

- Breakpoints are supported for debugging the execution model by bringing up the debugger just prior to the point where the step is to be executed.
- The heavy lifting is done by the executionStep itself in its **executeIn:** method. The knowledge about what to do is embodied in the executionSteps themselves.

Executing a step

The behavior of most ExecutionSteps is quite simple. They push a qualifier onto the stack, pop it, etc. The SEMSend step code shown below is one of the more complex.

```
SEMSend>>
executeIn: executionModel
   "<executionModel: hierarchyOf SEMExecutionModel, ^self>"
   "Direct the execution model to take the steps needed for the message
    send.  If a disqualification occurs, see if making qualifier(s) more
    concrete will solve the problem."

   self receiver isDisqualified
      ifTrue: [^self disqualifiedReceiverIn: executionModel].
   self getResultQualifier.
   (self hasDisqualification or: [self result isNone])
      ifTrue: [self tryMoreConcreteExecutionIn: executionModel].
   self hasDisqualification
      ifTrue: [self notify: executionModel].
   executionModel privacyChecking
      ifTrue: [self checkForPrivacyViolationIn: executionModel].
   executionModel push: self result
```

Notes:

- If the receiver qualifier is disqualified, execution cannot proceed and the executionModel is notified.
- The real work is done by self getResultQualifier (see below).
- The code ‘self tryMoreConcreteExecutionIn:’ handles the case where the current method is abstract, i.e., it is intended to be invoked for instances of subclasses of the abstract class in which it is defined. In that case, some of the messages it sends to ‘self’ may not be defined in the abstract class. ‘tryMoreConcreteExecutionIn:’ looks for implementations in the subclasses.
- After the message send has been processed, it checks to see if any alternatives have disqualified. If so, it stores the first disqualification in the executionModel. Note however, the send step itself remembers all disqualifications so no information is lost.
- If privacy checking is desired, the ‘checkForPrivacyViolationsIn:’ method looks for cases where message sends have violated privacy rules.

Here is the code for the two key methods: the first is ‘getResultQualifier’ which in turn sends ‘getResultQualifierForReceiver:’ to each relevant signature.

```
getResultQualifier
   "<^self>"
   "Get return quals from signature(s) and put in result.  In the
    process check for and save partial or complete disqualification."

   self signatures size > 1
      ifTrue: [self getResultQualifierForAlternativesReceiver]
      ifFalse: [self getResultQualifierForReceiver: self signatures first]


getResultQualifierForReceiver: aSignature
   "<aSignature: hOf SEMSignature, ^self>"
   "Get return quals from signature and put in result.  In the
    process check for and save partial or complete disqualification."

   | resultQualifier |
   resultQualifier := SEMAlternativesQualifier new.
   self addResultQualifierForReceiver: aSignature to: resultQualifier.
   self result: resultQualifier reduced
```

Notes:

- We add the return qualifier from each qualified signature to the alternatives qualifier for the result of this message send.
- At the end, we reduce the alternative qualifier result, i.e., remove duplicates and convert it to a singleton if only one alternative remains.

The ExecutionModel

Class structure

Object subclass: #SEMExecutionModel

```
    instanceVariableNames: 'attributes qualStack contextStack externalStack
      lastPopped conditionalBlockDepth notifications returns
      methodArtifact semSignature privacyChecking visibleStepCount '
    classVariableNames: ''
    poolDictionaries: 'SEMExceptions '
```

- attributes – iOf Dictionary {key: hOf String, value: hOf SEMAttribute}, collection of all attributes in the scope of the method that are accessed during the analysis, e.g., args, temps, instVars, etc. Note that attributes inside blocks, e.g., block args and block temps, are managed by the block's executionContex, not by the method.
- qualStack – hOf OrderedCollection {of: hOf SEMQualifier}, the stack of object qualifiers uses during the analysis
- conditionalBlockDepth – hOf Integer, a count of how deeply the current executionStep is nested in conditional blocks.
- contextStack – hOf OrderedCollection {of: hOf SEMExecutionContext}, the stack of

contexts invoked during this method analysis. As the method is entered its executionContext is pushed. As literal blocks are evaluated their context is pushed. When a literal block exits, its context is popped. The next executionStep to be evaluated is always the one pointed to by the executionContext on the top of the context stack.

- externalStack – hOf OrderedCollection {of: hOf SEMQualifier}, the stack as seen by the sender of the message that would have invoked this method. This is intended to support future ability to step into methods during the static analysis.
- lastPopped – hOf OrderedCollection {of: hOf SEMQualifier}, intended for allowing undo which is not implemented yet.
- methodArtifact – hOf NAFMethodArtifact, the proxy for the method under analysis
- notifications – iOf OrderedCollection {of: hOf SEMNotification}, collection of disqualifications, warnings, and informational notes encountered in the static analysis.
- privacyChecking – (true | false), a flag to trigger checking for privacy violations if desired
- qualStack – hOf OrderedCollection {of: hOf SEMQualifier}, the stack of qualifiers being pushed and popped during the execution.
- returns – hOf OrderedCollection {of: hOf SEMQualifier}, collection of return qualifiers from the method (i.e., a method can have multiple returns)
- semSignature – hOf SEMSignature, the signature for the present method
- visibleStepCount – hierarchyOf Integer, a counter of steps that map visibly to the code (i.e., excluding housekeeping steps such as dupeTopOfStack). This variable supports visualization of the execution.

Key responsibilities and methods

Stepping through the static analysis

- *nextStep* – <^hierarchyOf SEMExecutionStep> Execute and return the next step. See method presented previously. If it has a breakpoint set, halt.
- *stepToEnd* – <^(true | false)> Repeatedly step until end or disqualification. Return true if reaches end OK, false if a disqualification occurs.

Push and pop stacks

- *pop* – <^hierarchyOf SEMQualifier> Remove and return the Qualifier on top of the qualifier stack. Also push it onto the 'lastPopped' stack in case we need to retrace the execution.
- *popContext* – <^hierarchyOf SEMExecutionContext> Pop and store the top of the qualifier stack (the result of the block evaluation) into the executionContext's resultQualifier to be used as the return qualifier of the literal block context. Then remove the top context from the context stack and return it.
- *push: aQualifier* – <aQualifier: hierarchyOf SEMQualifier, ^arg1> Put aQualifier on top of stack (at end of collection)

- *pushContext: anExecutionContext* – <anExecutionContext: hierarchyOf SEMExecutionContext, ^arg1> Put anExecutionContext on top of context stack (at end of collection). Note: This is equivalent to starting the execution of the context since the next step to be executed will now come from the new context

Fetch attributes

- *addReturn: aQualifier* – <aQualifier: hierarchyOf SEMQualifier, ^hierarchyOf SEMQualifier> Add qualifier to the returns collection if it qualifies. Add disqualifier if not. Reduce it and remove 'none' if necessary, and update the return attribute. Return the added qualifier
- *addSignatureAttributes* – <^self> Resolve and install the methodArg and return attributes in my attributes dictionary keyed by their variable names. Replace method arg qualifiers with instance qualifiers that know their arg position. The return qualifier has references to self and argN replaced with the appropriate instance qualifiers
- *attributeForVariableNamed: aString* – <aString: hOf String, ^(hOf SEMAttribute | nil)> Get the attribute for the named variable
- *resolvedSelf* – <^hierarchyOf SEMQualifier> Create and return a qualifier for self, i.e., a qualifier with the class set to the class in which this method is implemented.

ExecutionContext

Class structure

Object subclass: #SEMExecutionContext

```
    instanceVariableNames: 'executionModel steps stepPointer attributes
      resultQualifier hasBeenEvaluated '
    classVariableNames: ''
    poolDictionaries: 'SEMExceptions '
```

- attributes – iOf Dictionary {key: hOf String, value: hOf SEMAttribute}, attributes private to the context, e.g., block temps, block args.
- executionModel – hierarchyOf SEMExecutionMode, the execution model in which this context is executing.
- hasBeenEvaluated – (true | false), flag to record the fact that the SEM has stepped through the block.
- resultQualifier – (hierarchyOf SEMQualifier | nil), is nil if execution hasn't finished.
- stepPointer – hierarchyOf Integer, index of the last step executed
- steps – hierarchyOf OrderedCollection {of: hierarchyOf SEMExecutionStep}, the steps for the code in the method or block represented by the context.

Key responsibilities and methods

Accumulating steps

- *AddMethodEntry* – <^self>, create and add an EnterMethodContext step. Its referent is the self token for the executionModel.
- *AddMethodExit* – <^self>, create and add an ExitMethodContext step. Its referent is the executionModel.
- *addDefaultSelfReturn* – <^self>, add default return of self – last step for methods without a return in last step.

Managing attributes

- *addAttributeFor: aToken* – <aToken: hOf SEMToken, ^hOf SEMAttribute>, create and install a new attribute for aToken. Note: senders must ensure that the token type is either argument or temporary, the only attributes managed by an ExecutionContext.
- *attributeFor: aToken* – <aToken: hOf SEMToken, ^(hOf SEMAttribute | nil)>, answer the attribute for aToken if I have it, else answer nil.

Stepping through the static analysis

- *nextStep* – <^hierarchyOf SEMExecutionStep> Return next step. If context is complete return last step

Signature

Class structure

Object subclass: #SEMSignature

```
    instanceVariableNames: 'scanner context argQualifiers returnQualifier
      numberOfArgs positions '
    classVariableNames: 'SignatureCache '
    poolDictionaries: 'SEMExceptions NAFParsingExceptions '
```

- scanner – hOf NAFScanner, a simple token scanner that keeps the input string that created the signature
- context – hierarchyOf NAFMethodArtifact, the method artifact from which arg names can be obtained
- argQualifiers – hierarchyOf AbtOrderedDictionary {key: hOf String, value: hOf SEMQualifier}, the qualifiers in the signature keyed by their name. '^' is used as the key of the return qualifier.
- numberOfArgs – hierarchyOf Integer, the number of args in the signature.positions: instanceOf Dictionary {key: (iOf String | iOf Integer), value: hOf Integer},
- returnQualifier – hOf SEMReturnQualifier, the return qualifier

- SignatureCache – iOf Dictionary {key: (hOf Class | hOf Metaclass), value: iOf Dictionary {key: iOf Symbol, value: hOf SEMSignature}}, class variable that caches signatures for methods that do not contain their own signature. It is a dictionary keyed by the class with values that are dictionaries keyed by the selector symbol.

Key responsibilities and methods

Collect and provide qualifiers for arg and return attributes

- *addMethodArgumentQualifier: qualifier for: keyword* – <qualifier: hOf SEMQualifier, keyword: hOf String, ^self> Add variable-qualifier pair to dictionary.
- *addReturnQualifier: qualifier* – <qualifier: hOf SEMReturnQualifier, ^self> Add the given qualifier as the return qualifier."
- *qualifierFor: keyword* – <keyword: hOf String, ^ hierarchyOf SEMQualifier> Return a copy of the requested qualifier.
- *qualifierAtIndex: index* – <index: hOf Integer, ^ hierarchyOf SEMQualifier> Return the indexed arg qualifier.
- *returnAttribute* – <^hierarchyOf SEMAttribute> Return my return attribute. Note: a return attribute begins with an empty alternatives qualifier as the deduced qualifier.

Important subclasses

Two subclasses of Signature are used to return the result from a method lookup:

SEMSignature subclass: #SEMSignatureWithReceiver

```
instVars: 'clientSend receivingClass receivingQualifier '
```

- clientSend – hOf SEMSend, the Send step that invokes the signature's method.
- receivingClass – (hOf Class | hOf Metaclass), the class of the receiver, which may differ from the class that implements the method.
- receivingQualifier – hOf SEMQualifier, the qualifier of the reiceiver.

SEMSignatureWithReceiver ties together the signature found in the lookup, the send step that requested the lookup, the class of the receiver, and the qualifier of the receiver. Note, if the receiving qualifier is an alternatives qualifier, there may be more than one signature found.

SEMSignature subclass: #SEMNotUnderstood

```
instVars: 'enclosingClass selectorSymbol '
```

- enclosingClass – (hOf Class | hOf Metaclass)
- selectorSymbol – iOf Symbol

SEMNotUnderstood is returned when a method lookup fails. Its primary role is to provide information about the error.

Attribute

Class structure

```
Object subclass: #SEMAttribute
    instanceVariableNames: 'token deducedQualifier declaredQualifier '
    classVariableNames: 'ClassAttributeCache '
    poolDictionaries: 'SEMExceptions NAFParsingExceptions '
```

- token – hierarchyOf SEMToken, the token for the variable modeled by this attribute. Note, the token encodes both the name and the variable type (e.g., argument, temporary, instVar, ...)
- deducedQualifier – hierarchyOf SEMQualifier, the qualifier inferred by the static analysis
- declaredQualifier – hierarchyOf SEMQualifier, the qualifier declared in the design
- ClassAttributeCache – iOf Dictionary {key: (hOf Class | hOf Metaclass), value: iOf Dictionary {key: iOf String, value: hOf SEMQualifier}}, holds the attributes for kernel classes. It is a class variable which is a dictionary keyed by the class with values that are other dictionaries keyed by the attribute name with values that are attribute qualifiers for the instance and class variables.

Key responsibilities and methods

Resolve unresolved qualifiers

- *resolveInExecutionModel: executionModel* – <executionModel: hOf SEMExecutionModel, ^self> If needed, replace instances of SEMSelf and SEMArgN with resolved instanceQualifiers.

Qualifier

Class structure

Object subclass: #SEMQualifier
  classInstanceVariableNames: 'parsingPrefixes partialParsingPrefixes '
  instanceVariableNames: 'isFuzzy '
  classVariableNames: ' '
  poolDictionaries: 'SEMExceptions NAFParsingExceptions '

The state maintained by SEMQualifier is related to qualifier's self parsing behavior. Qualifiers parse themselves from strings, e.g., the string defining a signature. However we are not interested in the parsing behavior for the purposes of this report.

Key responsibilities and methods

Method lookup

Following is the method lookup code in SEMQualifier:

```
signatureForSelector: aSelector sentTo: aClass
    "<aSelector: hOf Symbol, aClass: (hOf Class | hOf Metaclass),
     ^(hOf SEMSignatureWithReceiver | hOf SEMNotUnderstood | hOf SEMSignature)>"
    " Models virtual machine method lookup, returning the signature of
     the receiver."

    | receiver signature |
    receiver := aClass.
    [(receiver notNil and: [receiver methodDictionary includesKey: aSelector])
       ifTrue:
          [signature := self signatureForReceiver: receiver selector: aSelector.
           signature isUnderstood
             ifTrue:
                [^SEMSignatureWithReceiver from: signature
                                        forReceivingClass: aClass
                                        receivingQualifier: self]].
     receiver == nil]
       whileFalse:
          [receiver := receiver superclass].
    ^self enclosingClass: aClass doesNotUnderstand: aSelector
```

Qualification

- *qualifyingClasses*, Answer a collection of classes that satisfy the receiver's qualification. For the abstract Qualifier class, it returns an empty set. Subclasses do the appropriate thing.
- *qualifies: aQualifier* – <aQualifier: hierarchyOf SEMQualifier, ^(true | false)>, subclass implementors of this message answer the question: is the set of objects that qualify for aQualifier a subset of those that qualify for me.
- *partiallyQualifies: aQualifier* – <aQualifier: hierarchyOf SEMQualifier, ^(true | false)>, are some of aQualifier's alternatives qualified by me?

Resolution of qualifiers in the context of an execution model or signature

- *resolvedIn: executionModel* – <executionModel: hOf SEMExecutionModel, ^self>, the abstract qualifier returns self which is already resolved. Subclasses that represent unresolved qualifiers, such as SEMSelf and SEMArgN, do the appropriate resolution.
- *resolvedInSignature: signature* – <signature: hOf SEMSignatureWithReceiver, ^hierarchyOf SEMQualifier>, default is to return self which is already resolved. Subclasses do appropriate resolution. For example, SEMSelf returns the receiver qualifier of the send. SEMArgN returns the appropriate arg qualifier passes to the message send.
- *resolvedInReceiverOf: aSend* – <aSend: hOf SEMSend, ^self>, default is to return self which is already resolved. SEMSelf returns the receiving qualifier of the send. SEMArgNAspect refers to an aspect of one of the args to the method and returns that aspect. And so forth.

Behavior of subclasses

- AlternativesQualifier – maintains its qualifiers, iOf OrderedCollection {of: SEMQualifier}. Its *qualifyingClasses* method answers the set of qualifying classes implied by all its qualifiers.
- BlockQualifier – maintains its return qualifier and any arg qualifiers. The *qualifyingClasses* method answers the 'Context' class. Compares itself with a blockSignatureQualifier via the method – blockSignatureQualifies: aBlockQualifier which checks that its block signature is appropriate for aBlockQualifier. To do so, it follows the rule of contravariance, i.e., answers true if aBlockQualifier's args are more general than its own and aBlockQualifier's return is a subset of its own.
- ClassQualifier – important instVars are: rootClass (iOf Symbol) which is the name of the class it represents, and meta (true | false) which is true if the qualifier represents the class itself, false if it represents an instance.
- Disqualifier – This qualifier marks a disqualification, i.e., a mismatch. Its primary behavior is to answer false to the *qualifies:* message. It also holds an instVar, reason (hOf String), that keeps an explanation for the disqualification.
- InstanceQualifier – created when an unresolved qualifier such as self is resolved. Important isntVars are: pseudoVar (iOf String) which specifies the pseudoVar that gave rise to this instanceQualifier (e.g,, self, arg1, …), qualifier (hOf SEMQualifier) which is the resolved qualifier, and copy (true | false) which is true if this arose from a qualifier like 'self copy'. When an instanceQualifier is asked if it qualifies another qualifier, it answers true only if the other qualifier is an instanceQualifier for the same pseudoVar with the same copy status. For other issues of qualification, it delegates to the actual qualifier it holds.
- LiteralBlockQualifier – instVars are: executionContext (hOf SEMExecutionContext) that holds the executionSteps for the literal block, numberOfArgs (hOf Integer), and temporaries (hOf OrderedCollection {of: hOf EsLocal}) which specify any temp variables defined within the block.
- Any – a qualifier that qualifies any other qualifier that is not disqualified. It responds to the *qualifyingClasses* message with a collection containing Object and any other subclasses of nil in the system.
- False, Nil, True – qualifiers that represent the unique instances. They respond to *qualifyingClasses* in the obvious way. The respond to *qualifies: aQualifier* by answering *^aQualifier = self*.
- None – used for a return qualifier where the method or block does not return.. The only qualifier it qualifies is another instance of *None*.
- Self, ArgN, SelfAspect, ArgNAspect, SelfCopy, ArgNCopy – unresolved qualifiers that appear in signatures but do not participate in the operation of the SEM until they have been resolved. The ArgN varieties of these qualifiers know their index, i.e., which arg they represent. The three key methods implemented by these qualifiers are: *resolvedIn: executionMode,* which returns the appropriate qualifier from the method under analysis, *resolvedInReceiverOf: aSend* and *resolvedInSignature: aSignatureWithReceiver,* both of which return the appropriate qualifier from the receiver of the send (the choice between

the last two is simply a matter of which object is handy at the moment, the send, or the signature).

- SelfValue, ArgNValue – unresolved qualifiers that apply only to blocks. They refer to the result qualifier obtained by sending the *evaluationResult* message to the block. This causes the evaluation of the block if it has not already been evaluated..
- Unspecified – a qualifier that plays the special role of marking the fact that a newly created object requires an aspect that must be deduced by usage in the method. This is usually due to its assignment to a temp variable that has no declared qualifier. The two key behaviors, which are implemented by SEMQualifierAspect, are: *adoptAspectSpecifiedBy: specifiedAspect*, and *adoptAspectFrom: aQualifier forKey: aspectKey*. Both of these cause the adopted aspect to be added to the alternatives qualifier that includes the unspecified qualifier.

QualifierAspect

Class structure

```
Object subclass: #SEMQualifierAspect
    instanceVariableNames: 'aspects '
    classVariableNames: 'AspectCache DefaultAspects '
    poolDictionaries: 'SEMExceptions '
```

- aspects – instanceOf LookupTable {key: hOf Symbol, value: hOf SEMQualifier}, the key value pairs of the aspect(s) of the qualifier this aspect modifies.
- DefaultAspects – iOf Dictionary {key: Symbol, value: Symbol}, a dictionary that specifies the default aspect *keys* of common classes. For all the simple collections, e.g., Array, OrderedCollection, String, Symbol, Set, etc., the default aspect key is *of:*. For streams, the default is *on:*. And for the various kinds of dictionary it is *value:*.
- AspectCache – iOf Dictionary {key: iOf Symbol, value: iOf Dictionary {key: iOf Symbol, value: hOf SEMQualifier}}. The top level dictionary is keyed by className symbols, its values are dictionaries that are themselves keyed by aspect symbol (e.g., #of:) with values that are the qualifier to be used as the default aspect for that key in that class. That is, this cache holds the default *qualifiers* for the various aspects used by the common classes. In most cases, including the simple collections, the default of: qualifier is any. Because this is so uninformative, it is seldom useful to take the default. But others are usually correct, e.g., ByteArray's default is hOf Integer, String's default is hOf Character, Interval's is hOf Integer, Stream's *on:* aspect is hOf String and its *of:* aspect is hOf Character. In these cases, declaring the aspect is a matter of taste.

Key responsibilities and methods

Qualification

- *qualifies: aQualifierAspect* – <aQualifierAspect: hierarchyOf SEMQualifierAspect, ^(true | false)>, answer true if the set of the aspect/qualifier pairs in aQualifierAspect is a

subset of mine.

Resolution

- *adoptAspectFrom: aQualifier forKey: aspectKey* – <aQualifier: hOf SEMClassQualifier, aspectKey: iOf Symbol, ^self>, I have an unspecified aspect. Add aQualifier as an alternative along with the unspecified aspect.
- *asSpecifiedBy: specifiedAspect* – <specifiedAspect: hOf SEMQualifierAspect, ^hOf SEMQualifierAspect>, I have an unspecified aspect. Answer a copy of myself that has taken on the specification defined by the corresponding aspect in aQualifier.
- *RemoveNoneAndUnspecified* – <^self>, remove *none* and *unspecified* from my aspect(s).
- *resolveAspectsFor: aSend* – <aSend: hierarchyOf SEMSend, ^self>, Resolve any unresolved aspects from the send, i.e., iterate over my aspects, replacing each unresolved aspect with the result of sending it the message resolvedInReceiverOf: aSend.

ExecutionStep

Class structure

```
Object subclass: #SEMExecutionStep
    instanceVariableNames: 'start end referent result breakpoint '
    classVariableNames: ''
    poolDictionaries: 'SEMExceptions '
```

- start – hOf Integer, the position of the first character in the code that gives rise to this step.
- end – hOf Integer, the position of the last character in the code that gives rise to this step.
- referent – (hOf SEMToken | hOf EsLiteral | hOf SEMQualifier | hOf SEMExecutionModel)
- result – hOf SEMQualifier. Note: for most steps the result is a qualifier. But it has become a grab bag of misc things for some steps, e.g., the result of a EnterConditionalBlock is an integer. These misc results are to support the needs of tools like the OID generator that need odd bits of information, especially from housekeeping steps that start and end blocks.
- breakpoint – (true | false), true if the execution should halt and bring up a debugger.

Class hierarchy

Following are the subclasses of SEMExecutionStep. Indenting indicates subclassing.

```
SEMExecutionStep instVars: 'start end referent result'
   SEMConvertToBlockQualifier
   SEMDupeTOS
   SEMEnterContext
      SEMEnterBlockContext
         SEMEnterConditionalBlock instVars: 'keyword'
      SEMEnterMethodContext
   SEMExitContext
      SEMExitBlockContext
         SEMExitConditionalBlock
      SEMExitMethodContext
   SEMMergeTopTwo instVars: 'enterBlock1 enterBlock2 result1 result2'
   SEMPopAndStoreBlockArg
   SEMPopAndStoreVar
   SEMPopForSend instVars: 'send'
      SEMPopArg instVars: 'arg argIndex'
      SEMPopReceiver
   SEMPopTOS
   SEMPushImmediate instVars: 'qualifier'
   SEMPushLiteral
   SEMPushLiteralBlock
   SEMPushVar
   SEMRepush
   SEMReturnImmediate
   SEMReturnTOS instVars: 'rawReturnQualifier'
   SEMSend instVars: 'numberOfArgs args signatures receiver
                                    disqualifications'
   SEMStoreVar
```

Key abstract responsibility

- executeIn: anExecutionModel – <executionModel: hierarchyOf SEMExecutionModel, ^self>. Each subclass implements this method. These methods collaborate with the executionModel to accomplish the desired behavior of the particular step. That is, they send messages to the executionModel to push and pop stacks, fetch from and store to attributes, and return results. They also collaborate with the object in their 'referent' instVar which contains the element of code (e.g., a token, a literal, or a qualifier) that the step refers to, if any.

Behavior of subclasses

- ConvertToBlockQualifier – Generated by EsWhileStatement. Used only for receiver of a whileTrue: or whileFalse: message. Converts the top item on the stack into a zero arg block qualifier with its return set to the item on the top of the stack.
- DupeTOS – Generated by EsAssignmentExpression and EsCascadedExpression. Pushes a duplicate of the top qualifier on the stack.
- EnterBlockContext – Generated by EsBlock. Requests the LiteralBlockQualifier to set up attributes for any local temp variables.
- EnterConditionalBlock – Generated by EsBlock when the block is an argument to messages like #ifTrue:, #ifFalse:, etc. Sets up local attributes and increments the conditionalBlockDepth on entry to the block. If the conditional test that controls the

block's execution can be used to deduce more about an attribute, it does so and adds the newly deduced attribute qualifier to the local attributes.

- EnterMethodContext – Generated by EsMethod. Tells the executionModel to get self and args from the externalStack.
- ExitBlockContext – Generated by EsBlock. Saves the result of the execution of the block and then pops the literal block qualifier off of the stack.
- ExitConditionalBlock – Generated by EsBlock. Does same as ExitBlockContext, and in addition, decrements the conditionalBlockDepth.
- ExitMethodContext – Generated by EsMethod. Saves the result of the method execution.
- MergeTopTwo – Generated by EsBlock. Pop the top two stack items, combine them into an AlternativesQualifier, and push result. Used for result of #ifTrue:ifFalse:.
- PopAndStoreBlockArg – Generated by EsTemporaries.
- PopAndStoreVar – Generated by EsAssignmentExpression. Store the qualifier on the top of the stack to the attribute for the variable. Then pop the stack if the store succeeded (i.e., if there was no qualifier mismatch). If the store is disqualified, leave the result on top of the stack.
- PopArg – Generated by EsCascadedExpression, EsMessageExpression, and EsWhileStatement. Pops top qualifier and saves it, resolves it if necessary, and passes it to the Send step. If the arg is a literalBlock that must be evaluated, initiate the evaluation.
- PopReceiver – Generated by EsCascadedExpression, EsMessageExpression, and EsWhileStatement. Pops top qualifier and saves it. If the receiver is a block that should be evaluated, initiate the evaluation.
- PopTOS – Generated by EsCascadedExpression and EsStatement. Simply pops the top qualifier. The PopTOS step actually keeps the popped qualifier "for the record" but the effect on execution is that it is discarded. This is necessary for intermediate expression results in a cascade, and for the result of most statements.
- PushImmediate – Generated by EsBlock. Push a specific qualifier stored with the step. Used for housekeeping to push a SEMNone qualifier to be returned if the block ends with a return from the method.
- PushLiteral – Generated by EsLiteral. Pushes the qualifier appropriate to the literal (e.g., hOf Integer, or hOf String).
- PushLiteralBlock – Generated by EsBlock. Pushes the LiteralBlockQualifier.
- PushVar – Generated by EsBlock, EsLocalReference, and EsVariable. Pushes the deducedQualifier from the attribute that models the variable, if one has been deduced, else push the declaredQualifier.
- Repush – Generated by EsBlock. Pushes the last qualifier popped from the stack. Used to replace the value of the last statement executed in a block (which is popped routinely at the end of each statement) so that it becomes the value of the block.
- ReturnImmediate – Generated by EsMethod. Used to return the default self qualifier at the end of a method.

- ReturnTOS – Generated by EsStatement. Used for “return” statements.
- Send – Generated by EsCascadedExpression, EsMessageExpression, and EsWhileStatement. Its function has been described earlier.
- StoreVar – Not used by current SEM. Stores qualifier to deducedQualifier of appropriate attribute.

Generation of ExecutionSteps

From code

Each node of the parse tree implements an #addStepsTo: method. The details of how this is done depend upon specifics of the parse tree as well as upon the execution steps. Below are some representative examples from the VisualAge ParseNodes.

```
EsStatement
addStepsTo: aContext
   "<aContext: hOf SEMExecutionContext, ^self>"
   "Add the primitive virtual machine operations used to execute
    the statement to aContext."

   self expression addStepsTo: aContext.
   self isReturn
       ifTrue:      "return object on top of stack (deleting it)"
           [aContext addStep: (SEMReturnTOS start: sourceStart
                       end: self sourceEnd
                       referent: self)]
       ifFalse:     "discard object on top of stack"
       [aContext addStep: SEMPopTOS new]
```

Note: if the statement is not a return statement, its value, which is left on the top of the stack, must be discarded.

```
EsAssignmentExpression
addStepsTo: aContext
   "<aContext: hOf SEMExecutionContext, ^self>"
   "Add the primitive virtual machine operations used to execute
    the Expression to aContext."

   self rhs addStepsTo: aContext.
   aContext addStep: SEMDupeTOS new.
   aContext addStep: (SEMPopAndStoreVar start: variable sourceStart
                       end: variable sourceEnd
                       referent: variable variableToken).
```

Note: we duplicate the top of stack qualifier so that one copy will remain as the value of the expression after the other is popped and stored in the variable.

```
EsLocalReference
addStepsTo: aContext
   "<aContext: hOf SEMExecutionContext, ^self>"
   "Add step to push the variable."

   aContext addStep: (SEMPushVar start: self sourceStart
                            end: self sourceEnd
                            referent: self variableToken)
EsBlock
addStepsTo: aContext
   "<aContext: hOf SEMExecutionContext,
     ^instanceOf SEMLiteralBlockQualifier>"
   "Create a new LiteralBlockQualifier with its new blockContext and
    add to it the execution steps needed to execute the block."

   | blockContext blockQualifier |
   blockContext := SEMExecutionContext forModel: aContext executionModel.
   blockQualifier := SEMLiteralBlockQualifier executionContext:
                        blockContext.
   blockQualifier numberOfArgs: self numberOfArgs.
   blockQualifier temporaries: self temporaries.
   blockContext addStep:
      (SEMEnterBlockContext start: self sourceStart end: self sourceEnd
                  referent: blockQualifier).
   self addOpenCodedStepsTo: blockContext.
   blockContext addStep:
      (SEMExitBlockContext start: self sourceStart end: self sourceEnd
                  referent: blockQualifier).
   aContext addStep:
      (SEMPushLiteralBlock start: self sourceStart end: self sourceEnd
                  referent: blockQualifier).
   ^blockQualifier
```

Note: this method creates the scaffolding for the block. The following method creates the innards of the block.

```
addOpenCodedStepsTo: aContext
   "<aContext: hOf SEMExecutionContext, ^self>"
   "Add the execution steps needed to execute the block.
    Pop args if any to #methodTemp or #blockTemp attributes
    statements - leave last result on stack as value of block.
    Issue: if last statement is return, block doesn't have value"

   self hasArgs
      ifTrue: [arguments addStepsTo: aContext].
               "Note: we avoid the arguments getter here,
                it doesn't return the instVar"
   self hasStatements
      ifTrue:
         [self addStatementStepsTo: aContext]
      ifFalse:        "put nil on the stack"
         [aContext addStep: (SEMPushVar
                           start: self sourceStart
                           end: self sourceEnd
                           referent: (SEMToken type: #pseudoVar
                                               textString: 'nil'))]
```

```
EsCascadedExpression
addStepsTo: aContext
   "<aContext: hOf SEMExecutionContext, ^self>"
   "Add steps for the CascadedExpression.  Push he receiver then for
    all but the last message, duplicate the stack top, push the
    args, push a #send directive,  and discard the stack top (send
    result).  For the last message, don't duplicate the stack top or
    discard the result."

   | lastMessage |
   lastMessage := self messagePatterns last.
   self receiver addStepsTo: aContext.
   self messagePatterns do: [:aMessage |
      lastMessage == aMessage
         ifFalse: [aContext addStep: SEMDupeTOS new].
      aMessage addStepsTo: aContext.  "message pushes args"
      self addSendStepsFor: aMessage to: aContext.
      lastMessage == aMessage
         ifFalse: [aContext addStep: SEMPopTOS new]]


EsKeywordPattern
addStepsTo: aContext
   "<aContext: hOf SEMExecutionContext, ^self>"
   "Add the primitive virtual machine operations used to push
    my args onto the stack.
    Note: the VM expects self to have been pushed first, then
       first arg, second arg, etc."

   self arguments do: [:anArg |
      anArg addStepsTo: aContext]
```

Example of execution steps generated for a simple method

Consider the following simple method:

```
numberOfButterfliesIn: butterflies
   "<butterflies: hOf Collection {of: hOf Butterfly}, ^hOf Integer>"
   "example of short method that returns the number of butterflies."

   | size |
   size := butterflies size.
   ^size
```

The ten execution steps generated for this method are as follows:

```
1. SEMEnterMethodContext
2. SEMPushVar (#argument butterflies)
3. SEMPopReceiver
4. SEMSend (#unary size)
5. SEMDupeTOS
6. SEMPopAndStoreVar (#temporary size)
7. SEMPopTOS
8. SEMPushVar (#temporary size)
9. SEMReturnTOS ^size
10.SEMExitMethodContext
```

Generation of execution steps from bytecodes

Because of the relatively close correspondence between execution steps and bytecodes, it should be possible to generate executionSteps directly from bytecodes without having to parse the source code. This could be important in assuring the accuracy of signatures for methods that have their source code hidden.

BIBLIOGRAPHY

[BG98a] S. L. Burbeck & S. G. Graham. A design virtual machine for static analysis of Smalltalk. IBM Technical Report TR29.3021, 1998.

[BG98b] S. L. Burbeck & S. G. Graham. Using signatures to improve Smalltalk productivity and reuse. IBM Technical Report TR29.3020, 1998.

[BI82] A. H. Borning and D. H. H. Ingalls. A type declaration and inference system for Smalltalk. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pp. 133 - 139, 1982.

[Bur96] S. L. Burbeck. Real-time complexity metrics for Smalltalk methods. IBM Systems Journal. Vol. 35, No. 2, pp. 204-226, 1996.

[GB98] S. G. Graham & S. L. Burbeck. Deriving object structure diagrams from Smalltalk code. IBM Technical Report TR29.3027, 1998.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80 The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.

[Ing83] Daniel H. H. Ingalls. The evolution of the Smalltalk virtual machine. In *Smalltalk-80 Bits of history words of advice.* Glenn Krasner, ed. pp. 9-28, Addison-Wesley, Reading, MA, 1983.

[JF88] R. E. Johnson and B. Foote. “Designing Reusable Classes,” Journal of Object-Oriented Programming, 1(2), 22-30,35, June/July, 1988.