From: L. Peter Deutsch, Mark S. Miller
To: Smalltalk implementors group
Subject: Proposal for 1987 Smalltalk Standard meta-level interfaces to object state

This is a working document proposing a complete definition of the standard protocols for meta-level access to object state and to executable code for the 1987 version of the Smalltalk-80 Virtual Machine. The proposal includes a clean separation of object-level and meta-level aspects of Smalltalk consistent with both security and meta-interpretive definition, and without a significant change in the style of the language or environment. In order to do so, we include a coherent design for transparent forwarding and non-methodical objects as well (note: the implementability of this proposal doesn't depend on actually implementing non-methodical objects, they are introduced for explanatory clarity). Comments are solicited. For changes in successive versions, consult the version history.

This document is the first of a pair specifying meta-level interfaces for the 1987 Smalltalk standard. It is meant to be read first. The second document proposes a similar definition for meta-level access to control contexts and to processes.

Acknowledgements:

We wish to credit several important sources for the ideas presented here concerning meta-level facilities:

Ehud Shapiro & FCP for demonstrating & providing a model of how meta-interpretation provides a semantics yielding both debuggability & security.

Patty Maes for an encapsulated model of reflection.

Danny Bobrow for the generalization of MemoryAreas into nestable SecurityAreas.

Henry Lieberman for the insight that inheritance can be done through message passing.

And finally, Carl Hewitt, for the conviction that message passing among encapsulated actors is sufficient for everything.

Version history:

[3] ** in progress **: first version (derived from exec87.text.2): major contributions from Mark Miller, providing a first attempt at consistency with a secure meta-interpretive story, also a first spec for transparent forwarding and non-methodical objects.

The execution mechanism of the Smalltalk-80 Virtual Machine is defined by a virtual instruction set and a standard set of objects for representing execution state, and an interpreter that operates on this instruction set and these objects. In place of this, we propose to define only the externally visible behavior of the execution mechanism, not a particular algorithm that executes a particular representation of code and uses a particular representation of dynamic execution state. Indeed, our approach even allows a single system to represent code and state in several different coexisting ways. In particular, we propose that the only entities in the system that know the representation of executable code (other than the compiler) be the classes of the code objects, and that the message protocol of these objects be the basis of portability for the rest of the system such as the debugger. We propose a similar arrangement for the representation of dynamic execution state (Context objects). In other words, we propose that each running Smalltalk system include:

- A VM foundation, not written in Smalltalk, whose detailed interfaces are not specified (except generically, in that it must eventually support executing programs written in the Smalltalk language, including all the primitive methods that form part of the language definition).
- A language implementation kernel, written in Smalltalk, consisting of some classes whose interfaces are partially specified, but whose implementations are not specified.
- A set of programming tools (e.g. Browser, Debugger, Explainer), written in Smalltalk, whose functionality is not part of the VM definition.

[3] We strongly encourage vendors to implement their tools using only the standard specified interfaces of the language implementation kernel.

Meta-level separation (new for [3])

Recent research in the Lisp and Logic Programming worlds has established the benefit of implementing debugging and monitoring functions in terms of a "meta-interpreter". The idea is that even though all the execution structures of the language can be represented within the language itself, there is an explicit distinction between USING these structures implicitly and operating ON the structures explicitly. For example, Brian Smith's Reflective Lisp systems provide a linguistic mechanism for shifting between levels at any time; Udi Shapiro's work on Concurrent Prolog establishes a fixed boundary between levels, which appears preferable for protection purposes.

In this document we propose a coherent separation between meta-functionality and the normal world of Smalltalk execution, using a style closer to that of Concurrent Prolog. We introduce two interacting types of meta-level access: access to structure, and access to process. Meta-level access to structure means access to the internal state of an object, as if from the point of view of an interpreter executing a method that can access the object's state directly. This allows a clean semantics for inspectors: ordinarily, access to an object can occur only through its external protocol, but an inspector must access the object's state in the same way as an interpreter. We reify this machinery by means of a MetaObject, whose external protocol gives access to the internal state of the object that the MetaObject holds.

In both Concurrent Logic Programming and Actor systems, the notion of object and process are unified, so this kind of meta-level access is sufficient for purposes of debugging. However, in Smalltalk, an object does not have its own thread of control. A Smalltalk process cannot be seen as simply a meta-interpreter which is implementing method execution for the objects it is interacting with, because multiple processes may need to view the same object from a meta-level. So just as we create MetaObjects to give controlled access to the state of an object, we view ExecutionContexts (a redesign of the current Smalltalk-80 Contexts) as the meta-level objects through which we can view process.

Note that we separate these two types of meta-level access deliberately: access to process implies access to structure, but access to structure can be given while denying access to process.

In order to talk about execution in a way which is cleanly related to meta-interpretation, we need to do clean meta-interpretation. In order to support the Smalltalk style of environment, objects being executed at one meta-level need to be able to talk to objects being executed at another. In order to accomplish this, we need to introduce some new language constructs to talk about the interpretation process. While these are motivated by explanatory purposes, the functionality they provide must exist at some level of any actual Smalltalk system that has the semantics we propose, and we therefore propose that they be included in the language.

Scope of definition

The classes whose protocol we are concerned with defining, and the general nature of the protocol which we will specify for each, are:

Structure:

- Access to state:
  - MetaObject
    - meta-level access to original object
  - SecurityArea
    - find metaObject for a given object
- Access to description:
  - ClassDescription
    - save and retrieve source code
    - compile from source to executable code
    - queries of various kinds
  - ExecutableMethod
    - locate the source
    - queries
- Message handling:
  - NonMethodicalObject
    - syntax & informal semantics
    - relation to blocks
  - ApparentForwarder
    - interaction with remote object
    - relation to metaObjects

Process:

- ExecutionContext
  - step, proceed, restart, return
  - get the method
  - follow the enclosure (home) links
  - examine and change the named variables
  - correlate execution point with source code
- Process
  - create, suspend, continue, kill
  - examine and change priority

In addition, we will be somewhat concerned with classes that interact with these, such as BlockClosure and Semaphore.

In the descriptions below, messages with no other specified return value return their receiver.

Object, MetaObject, MemoryArea, and SecurityArea (rewritten for [3])
==========

As noted above, the direct instance variable access messages instVarAt: and instVarAt:put: are not compatible with a robustly protected system. Instead, we take the view that we do not expect an object to provide a de-encapsulated view of itself; rather, we introduce the notion of a meta-object that provides a view of an object as if from the privileged point of view of an interpreter executing methods that operate on the object's state directly. We must also restrict the creation of meta-objects in a way that allows for protection. We choose to do this by reifying the notion of a "security area", which represents permission to deal with a particular set of objects from a meta-level view. It is only this security area which can convert a reference to an object into a reference to its meta-object. Given just a reference to an object, but without reference to its security area, we intend that there be no way to gain access to the meta-object.

<MetaObject | nil> := <SecurityArea> metaObjectFor: <Object>

A return value of nil means that this SecurityArea doesn't represent meta-level permission for this object.

The Smalltalk kernel only provides a flat space of SecurityAreas -- one for each physical object memory system. (An ordinary single-user Smalltalk system will probably have only one object memory system, or perhaps a second one for handling out-of-memory emergencies.) Note that the user may easily define new classes of SecurityAreas which are subsets or unions of those provided by the kernel.

<Object> := <MetaObject> subject

This message returns the original object that this meta-object is a meta-object for.

<Behavior> := <MetaObject> classOfSubject

Returns the actual class of <MetaObject subject>. Note that this is not necessarily the same as <subject class>, as <subject class> returns only what the object externally claims is its class. In particular, if we ask a transparent forwarder what its class is, we find the class of the object it is forwarding to. If we ask its meta-object what its classOfSubject is, we get NonMethodicalObject (to be introduced later).

<Dictionary> := <MetaObject> instVarDictionary

This message returns an object that appears to be a Dictionary, in which the keys are the original receiver object's instance variable names (if the object has named instance variables) or an appropriate range of integers (if the object has indexed instance variables). This dictionary responds to the full range of Dictionary protocol: thus, for example, it provides for finding the value of any instance variable (using at:), setting the value of any instance variable (using at:put:), enumerating the instance variables (using do: or keysDo: or associationsDo:), etc. Note that the dictionary continues to present the current state of the object, not a snapshot. Needless to say, this "dictionary" is not an instance of the standard Dictionary class, it merely implements Dictionary protocol.

We assume that an object's SecurityArea has unconstrained access to the object memory system that stores the state of the object. Therefore, an object's meta-object and instVarDictionary access (and change) this state by sending messages to the object's SecurityArea. We do not attempt, in this document, to standardize the protocol with which this interaction takes place.

One would like the Smalltalk "dependents" mechanism to be usable with meta-objects and instance variable access dictionaries, so that an object could be notified whenever the state of another object changed at the instance variable level. (The dependency mechanism is not part of the proposed standard, but is implemented in many existing Smalltalk systems.) While there is no logical obstacle to this, we recognize that implementing this may be difficult or impossible in some implementations of the Smalltalk-80 virtual machine. Therefore, we encourage the implementors of those systems that do provide a dependency mechanism to either implement it for meta-objects and instance variable access dictionaries, or to reject attempts to add dependents to such objects.

ClassDescription / Behavior

Unlike the present system, our proposal here does not make manifest any Compiler or Parser class. Instead, each class is responsible for compiling methods defined within it. This allows different classes to use different source languages.

Compiling

<ExecutableMethod> := <Behavior> compile: <String> in: <NameScope> ifError: <BlockClosure>
Compile a piece of source text and return an ExecutableMethod. If an error occurs, apply the BlockClosure to an "error object" argument. The error object must respond to at least the following messages:

string - return an error message string;
position - return a source position for indicating where the error occurred.
Implementations may choose to include further information in the error object.

Properties of ExecutableMethods are described in more detail below.

Accessing

The following messages from chapter 16 of the Blue Book are hereby standardized:
<Behavior> := <Behavior> superclass
<Behavior> superclass: <Behavior>
We would like to provide the complementary message
<(Collection of: Behavior)> := <Behavior> subclasses
Unfortunately, this is conceptually incompatible with distributed systems, systems in which only part of the code is loaded into the machine's address space (or fast retrieval space), etc. Another conceptual objection is that specializing a class should not result in a change in behavior of the class being specialized. The only solution we presently see for this problem is to provide, at the user-level (as opposed to the kernel, and therefore not part of this document), subclass registry dictionaries local to e.g. a specific user. When that user makes a subclass, he registers this with his local subclass registry. When he wants to find out what subclasses of a given class HE KNOWS ABOUT, he ask his local registry. We have not experimented with this idea, and it is clearly less convenient than the present arrangement.

The following messages are added or changed:
<(Collection of: Symbol)> := <Behavior> instVarNames
Return a Collection of the receiver's instance variable names. If the receiver has indexed instance variables, the result is a collection with one of the special symbols 'Object*' asSymbol or 'Byte*' asSymbol, compatible with the syntax for creating classes.
<Dictionary> := <Behavior>classVarDictionary
Returns a Dictionary of the receiver's class variables.
<Dictionary> := <Behavior> methodDictionary
Return a Dictionary-like object representing the methods known to this class. This object is probably not exactly a MethodDictionary object as in the current Smalltalk-80 system: for example, it may need to remember the class it came from, for selective cache invalidation when a new method is added. Again, the philosophy is to present a Dictionary-like interface, with the implementation left up to the system implementor. This interface replaces specialized accessing messages such as selectors (= methodDictionary keys), compiledMethodAt: (= methodDictionary at:), etc.

Note that all Behaviors, not just Classes, are specified as having instance and class variables.

We specifically do not propose to standardize any Behavior protocol regarding non-linguistic information such as organization categories, comments, subclass registries, etc.

Querying

Querying messages fall into two groups: those that return a set of methods with some property, and those that return a set of values obtained from all the methods in the class. The querying messages referring to variables only refer to those visible at the class level, i.e. the instance and class variables of the receiver class and its superclasses, plus the (read-only) pseudo-variables 'self' and 'super'.

<(Collection of: Symbol)> := <Behavior> whichMethodsRead: <Symbol>
<(Collection of: Symbol)> := <Behavior> whichMethodsWrite: <Symbol>
<(Collection of: Symbol)> := <Behavior> whichMethodsReadOrWrite: <Symbol>
Return the selectors of the methods which respectively read, write, or either read or write the

variable with the specified name. If there is no such variable visible from this class, return an empty collection.

<(Collection of: Symbol)> := <Behavior> whichMethodsSend: <Symbol>
Return the selectors of the methods which send the specified message. We explicitly require that this message, and its companion messagesSent, give accurate answers with reference to the source code: in particular, the answer must include messages sent at the source level that may not be explicitly present in the object code (such as ifTrue:), and must not include messages present in the object code that implement language features (such as blockCopy: in the present Smalltalk-80 system). Such exceptions might be handled with a small bit table in each ExecutableMethod, or with extra entries in a literal table, or in other ways. For performance reasons, we strongly discourage implementors from implementing this message (or any other query message) by reanalyzing the source code.

<(Collection of: Symbol)> := <Behavior> whichMethodsUseLiteral: <Object>
Return the selectors of the methods which refer to the specified object as a literal object. Symbols that are only used as selectors are NOT included in this query.

<(Collection of: Symbol)> := <Behavior> variablesRead
<(Collection of: Symbol)> := <Behavior> variablesWritten
<(Collection of: Symbol)> := <Behavior> variablesReadOrWritten
Return the names of the variables read, written, or either read or written by any method within the class.

<(Collection of: Symbol)> := <Behavior> messagesSent
Return the selectors of all messages sent by any method in this class.

<(Collection of: Object)> := <Behavior> literalsUsed
Return all the literals used by any method in this class.

ExecutableMethod

We explicitly take the position that every ExecutableMethod be able to access (or regenerate) its source code, and to report the class in which it was compiled and the selector it is associated with. (The latter follows from the ability to retrieve the source code.) The former capability is present in the current Smalltalk-80 system; the latter is not. We expect that most systems will store both pieces of information directly in the ExecutableMethod in some form. Note that we are not requiring that ExecutableMethods be the "inner" representation of compiled code in the Smalltalk system, only that they appear in the places specified by this document.

Accessing

<Behavior> := <ExecutableMethod> definingClass
Return the class where the method is defined.

<SmallInteger> := <ExecutableMethod> numArgs
Return the number of arguments that the method expects.

<Symbol | nil> := <ExecutableMethod> selector
Return the selector this method corresponds to, unless the method is the method of a non-methodical object (see below), in which case return nil.

<Object> := <ExecutableMethod> source
Return an object representing the source code for the method. The nature of the object is undefined: it need only be acceptable as the source argument in the compile:... method for the defining class. We do specify that in every system that includes a compiler for the standard source language, the compiler must accept Strings as input (i.e. objects that recognize String protocol and respond to the message asString by returning an appropriate String, such as String, Text, and Paragraph in the current Smalltalk-80 system.)

Querying

All the information-retrieval querying messages applicable to classes (but not the method-retrieval queries) are applicable to individual ExecutableMethods, namely:

<(Collection of: Symbol)> := <ExecutableMethod> variablesRead
<(Collection of: Symbol)> := <ExecutableMethod> variablesWritten
<(Collection of: Symbol)> := <ExecutableMethod> variablesReadOrWritten
<(Collection of: Symbol)> := <ExecutableMethod> messagesSent
<(Collection of: Object)> := <ExecutableMethod> literalsUsed

These queries only return the information obtained from the individual method. The visible variables are precisely those visible from within the method, i.e. those visible at the class level, plus the argument and temporary variables of the method (but not of any blocks), 'self', and the pseudo-variable 'super'.

We encourage implementors to implement these messages by returning a special kind of collection that extracts requested information from the ExecutableMethod on demand, rather than (for example) copying the entire response into an Array at the time the query message is received.

NonMethodicalObject

A non-methodical object is an object that can process with one method any message sent to it (as opposed to doing method lookup). It does exactly what objects that respond only to "doesNotUnderstand:" approximate. We propose the following Block-like syntax for creating non-methodical objects:

[::message | rest-of-a-block-after-the-parameters]

or, in the meta-syntax of the new syntax document:

non-methodical-object = '[' '::' declared-variable-name '|' [temporaries] statements ']'
primary = ... | non-methodical-object

Non-methodical objects are block-like closures, with all the same scoping rules as for blocks. They are also created in an identical fashion: by evaluating a non-methodical-object expression. The difference is in their response to messages. A non-methodical object responds to any message by binding the message itself to its one parameter variable and invoking its body.

The reason for introducing a syntax for creating non-methodical objects, rather than simply using the normal methods for creating an instance of class NonMethodicalObject, is that, like blocks, and unlike ordinary objects, they must be able to directly share variables already existing at the time the non-methodical object is created. (We hope that some future Smalltalk language will provide this ability for objects in general, but we don't have any proposal to make in this area at the moment.) Also, it must be possible to create and initialize a non-methodical object in a single atomic action, since one cannot send it initialization messages.

We also introduce the following syntax for sending computed messages:

<Object> :: <Object>

or, in the meta-syntax of the new syntax document:

keyword-message = ... | ('::' primary binary-message*)+

Note that '::' has the precedence of a keyword selector. The above expression does what "<Object> performMessage: <Message>" approximates: it sends the object on the right as a message to the object on the left. In any context and for any expr and body "[::x | body] :: expr" is equivalent to "[:x | body] value: expr". It is of course an error to send anything other than a <Message> to a normal, methodical

object. NonMethodicalObject and BlockClosure are both subclasses of Closure.

Again, since a non-methodical object must be able to handle EVERY message identically, we need a syntactic construct to express sending a message directly to an object: we cannot use a message for this purpose.

As NonMethodicalObjects do not perform method lookup in response to receiving a message, the class "NonMethodicalObject" should define no methods for its instances, and indeed must be distinguished syntactically in some way. A plausible implementation of NonMethodicalObjects as an incremental change to many systems is for the methodDictionary instance variable of the class NonMethodicalObject to contain something which is manifestly not an instance of MethodDictionary. Checking for this, which is only necessary when the various lookup caches fail and a full lookup is required, should not have significant performance impact.

ApparentForwarder
==========

An ApparentForwarder is typically the intermediary between a transparent forwarder and the object it is forwarding to. It responds to the following protocol:

```
<ApparentForwarder> := <ApparentForwarder class> subject: <Object>
<Object> := <ApparentForwarder> subject
<Object> := <ApparentForwarder> valueForMessage: <Object>
```

The third expression should be equivalent to:

```
<Object> := <ApparentForwarder> subject :: <Object>
```

A transparent forwarder can easily be set up to "front-end" for an ApparentForwarder as follows:

```
<ApparentForwarder> transparentForwarder
    ^ [ ::msg | self valueForMessage: msg ].
```

An apparent forwarder for an object which is actually on a remote machine cannot locally respond with the true subject, but it can respond to the "subject" message with a transparent forwarder which is front-ending for itself. This is why the above code uses "valueForMessage:" instead of "... subject :: ... " despite our stated behavioral equivalence. This equivalence only applies "on the outside". We apply this technique below:

MetaObject is a subclass of ApparentForwarder, with the subject being the object that the metaObject represents. It is consistent to view any object as just a transparent forwarder for its own metaObject, with all the work for responding to a message being contained the metaObject's valueForMessage: method. In some sense, the metaObject is forwarding the message to the idea of the object specified by its implementation, by bringing about the behavior specified by this implementation. In particular, the valueForMessage: method for a metaObject of a methodical object might be as follows:

```
<MetaMethodicalObject> methodForMessage: aMessage ifFail: failureBlock
    | selector class method |
    selector _ aMessage selector.
    class _ self classOfSubject.
    ^ class methodDictionary at: selector ifFail: failureBlock.

<MetaMethodicalObject> argsForMessage aMessage
    ^ aMessage args

<MetaObject> valueForMessage: aMessage
    | method args |
```

```
        method _ self methodForMessage: aMessage ifFail: [...].
        args _ self argsForMessage aMessage.
        ^ method valueForMetaReceiver: self arguments: args
```

An attempt to actually implement everything this way would cause one to infinitely regress up the infinite tower, but the system must act in a way consistent with this story. (It is allowed for the system to deviate from the story to the extent of executing within finite memory and time)

It is assumed above that the dictionary returned by "class methodDictionary" will search up the superclass chain in looking up a method. Note that since an ExecutableMethod knows its definingClass, it can process sends to 'super' without being told what class it was found in, e.g.:

```
    <ExecutableMethod> valueForMetaReceiver: aMetaObject sendSuper: aMessage
        | selector args class method |
        selector _ aMessage selector.
        args _ aMessage args.
        class _ self definingClass superclass.
        method _ class methodDictionary at: selector ifFail: [...].
        ^ method valueForMetaReceiver: self arguments: args
```

In order to implement the current Smalltalk semantics, the ifFail: blocks above should send doesNotUnderstand: messages to the original receiver. In order to address the "asArkObject" problem, we suggest that instead a "receiver: <original receiver> doesNotUnderstand: <message>" be sent to the selector. The method for this in class Symbol can in turn implement the current functionality.
}

Execution

```
[M-]{
        <Object> := <ExecutableMethod> valueForReceiver: <Object> arguments: <Array>
}
[M+]{
        <Object> := <ExecutableMethod> valueForMetaReceiver: <MetaObject> arguments: <Array>
}
```

Run the method with the given receiver and arguments, and return the resulting value. If [M-]{the object} [M+]{"<MetaObject> classOfSubject"} is not of a compatible class (i.e. the method's defining class or a sub...subclass), or the number of arguments is wrong, an error occurs. This message is provided mostly for 'doIt' in the interactive interface: BlockClosures are intended to be the objects normally used to represent executable procedures. Note that just as for any other send, there is no "unwind protection" in case a block does an ^ return. [M?]{I don't understand the comment about "unwind protection".}

[M+]{
Note that "arguments: <Array>" above is an <Array> of regular objects, not metaObjects. The <ExecutableMethod> needs priviledged access to the receiver, typically in order to "at:" and "at:put:" its instVarDictionary in the meta-interpretive story. However, it only needs to send messages to the other arguments. This is consistent with the story that the other objects are also being meta-interpreted in the same way: in this story these messages are just getting sent to transparent forwarders which are sending "valueForMessage:" messages to corresponding metaObjects which look up methods ...

By <MetaObject> and <ExecutableMethod>, we of course mean abstract data types corresponding to the protocols specified in this document. If both the <ExecutableMethod> and <MetaObject> above are implemented specially by the kernel, then this can execute without further messages being exchanged between them. (Indeed, this must be so to avoid infinite regress) However, if either is a non-special object which satisfies its abstract protocol, the other must interact with it properly according to this

protocol. This may be hard for primitive methods.