Externalizing Business-Object Behavior: A Point-and Click Rule Editor

Paul Davidowitz

EAGLE is an intergrated set of tools, architectures, processes, patterns, and reusable components that Andersen Consulting brings to its clients to enable the design and development of mission critical business systems. One of the frameworks explored in the area of tailorability was the externalization of business-object behavior, which makes conventional black-box business-object behavior available for tailoring by an end user. This externalized behavior is represented as rule-bases, which are specified in Smalltalk. This paper describes a tool that can be used to create and edit the rules of the rule-bases. The tool guarantees that the source code it produces will always compile and, at run time, will never generate the doesNotUnderstand: message. The tool is a context-sensitive editor that makes use of typing information and handles a wide range of expressions.

The tool guarantees that the source code it produces will always compile and, at run time, will never generate the **doesNotUnderstand:** *message.*

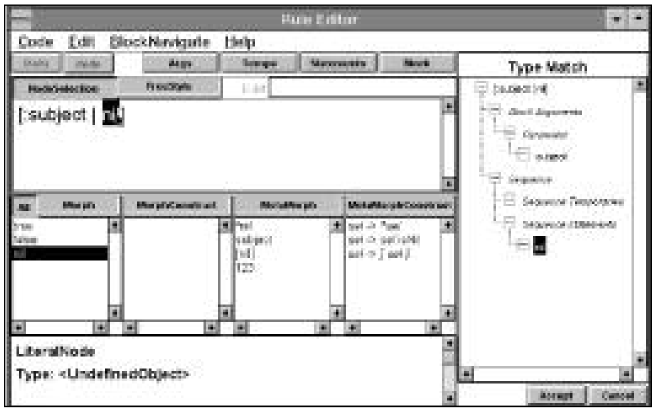
The basic technique is to transform a source string via manipulation of its abstract syntax tree. The abstract syntax parse tree, or more accurately ProgramNode tree in Visual Works, is produced as an intermediate step during code compilation, but is useful in its own right as an intermediate representation between source code and compiled code. The tree representation is convenient for dealing with syntactic issues. This representation can be transformed via decompilation into a string representation, with which most users prefer to interact. One representation can be transformed into the other as is convenient. Valid syntax and valid message-selectors are achieved by constraining the user to choose from valid manipulations of the ProgramNode tree. Valid message arguments are achieved by having the user choose from manipulations that satisfy type requirements.Here is an example of a rule-base consisting of two rules. The rule-base derives the attribute isReceiving for a Warehouse business object.

Rule1 premise: [:aWarehouse | aWarehouse isTakingInventory]
Rule1 action: [:aWarehouse | aWarehouse isReceiving: false]
Rule2 premise: [:aWarehouse | aWarehouse isTakingInventory not]
Rule2 action: [:aWarehouse | aWarehouse isReceiving: (aWarehouse isOpen and:[aWarehouse isAvailableStorage])]

As the example shows, a rule contains a single-argument block; the argument being the instance of the business object. The nature of Eagle rules is that they tend to be short and simple. The tool in turn was designed to perform well for shallow-nested blocks and a small number of assignments and temps.

PPD VisualWorks was the development environment; concepts may or may not be applicable to other Smalltalk environments.

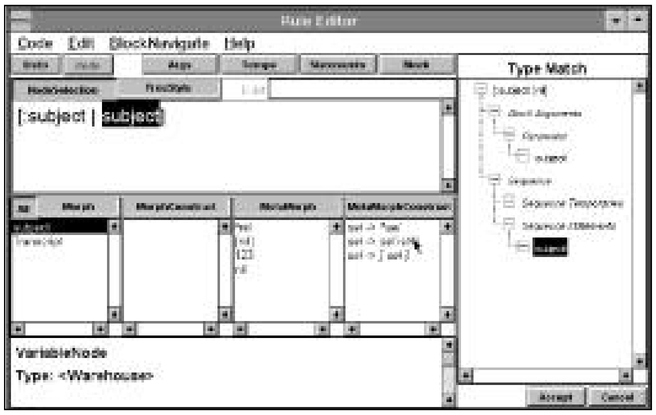
The standard input to the tool is a string specifying a single argument block and the class name of the business



```
[:subject | nil]
```

```
LiteralNode
Type: <UndefinedObject>
```

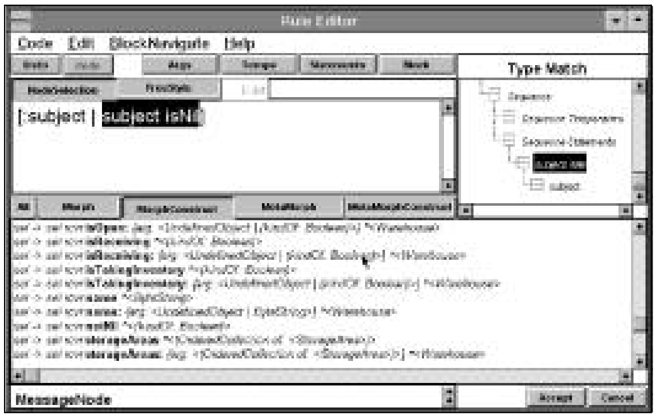
Figure 1. Starts with a generic seed-block of [:subject | nil].



```
[:subject | subject]
```

```
VariableNode
Type: <Warehouse>
```

Figure 2. DEMO 2.



```
[:subject | subject isNil]
```

```
sel -> sel rcvr isOpen: {arg: <UndefinedObject | (kindOf: Boolean)>} ^<Warehouse>
sel -> sel rcvr isReceiving ^<(kindOf: Boolean)>
sel -> sel rcvr isReceiving: {arg: <UndefinedObject | (kindOf: Boolean)>} ^<Warehouse>
sel -> sel rcvr isTakingInventory ^<(kindOf: Boolean)>
sel -> sel rcvr isTakingInventory: {arg: <UndefinedObject | (kindOf: Boolean)>} ^<Warehouse>
sel -> sel rcvr name ^<String>
sel -> sel rcvr name: {arg: <UndefinedObject | String>} ^<Warehouse>
sel -> sel rcvr notNil ^<(kindOf: Boolean)>
sel -> sel rcvr storageAreas ^<(OrderedCollection of: <StorageArea>)>
sel -> sel rcvr storageAreas: {arg: <(OrderedCollection of: <StorageArea>)>} ^<Warehouse>
```

```
MessageNode
```

Figure 3. DEMO 3.

object. The output is a similar string. Let's create the action block of Rule2, starting with a generic seed-block of [:subject | nil] as shown in Figure 1.

The view in the upper right is the ProgramNode tree. The user has the ability to select a node either by clicking on it in the tree view, or by clicking appropriately on code in the NodeSelection text view (shown). For example, to select a MessageNode in the text view, the user clicks on its selector. We click on nil, which is a LiteralNode.

We proceed to replace the nil statement with the argument of the block. We have replacement options available from four lists: Morph, MorphConstruct, MetaMorph, and MetaMorphConstruct. These options (collectively referred to as morphs) produce valid replacement nodes for the current-selection. Morph proper and MorphConstruct produce replacement nodes of the *same class* as that of the current selection; MetaMorph and MetaMorphConstruct produce those of a *different* class. The *construct* suffix means that the replacement node, instead of being a fixed prototype, is rather constructed from the current selection. For example, construct sel -> sel isNil (sel denoting current selection) means: replace the current selection with a MessageNode of selector isNil, and use the current selection as the receiver.

Let's metamorph. We select subject in the MetaMorph list. This replaces the LiteralNode with a VariableNode and, we get Figure 2.

The Morph list presents the option of using global Transcript; it is possible to have other globals as well. We select MetaMorphConstruct *sel -> sel* isNil. This replaces the VariableNode with a MessageNode, whose receiver is the VariableNode, and then we get Figure 3.

We have expanded the MorphConstruct list and see choices for different selectors. Each of these choices shows the return type (depicted by the up-arrow), as well as the required types for the arguments (if any). (The vertical bar appearing in a type specification is read as *or*.) We pick *sel -> sel rcvr* isReceiving:. This replaces the current MessageNode with another MessageNode of the same receiver, but different selector, that of isReceiving:.

Let's proceed by showing remaining steps with descriptive text, where bold emphasis indicates current selection.

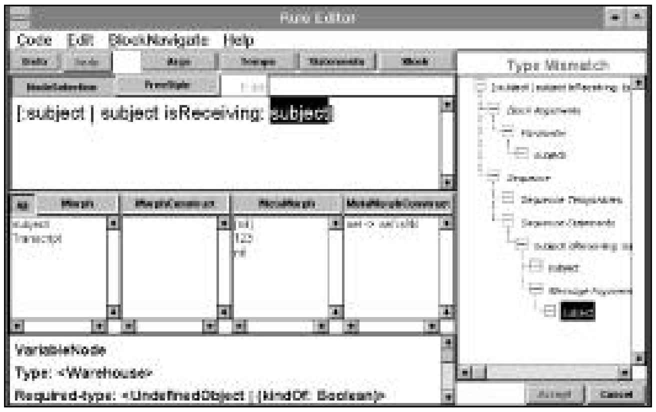
4) [:subject | **subject isReceiving: nil**]
5) [:subject | subject isReceiving: **nil**]
6) [:subject | subject isReceiving: **subject**]

At this point, as shown in Figure 4, the label at the top of the tree view has turned red and now reads: Type Mismatch instead of Type Match. The current selection appears (inverted) red in the tree view, and the Accept button at the lower right is disabled. The red color for the node indicates that for that particular node, there is a type mismatch. The type status in the lower left shows that the required-type is <UndefinedObject | (kindOf: Boolean)>, but the actual type is <Warehouse>. Warehouse is neither an UndefinedObject nor a kindOf: Boolean, so the required type is therefore not satisfied. Unless all nodes have their required type satisfied, the tool will not permit the code to be accepted.

Whereas each allowed manipulation of the ProgramNode tree will result in correct syntax, it will not necessarily satisfy required type for all ProgramNodes. As far as the tool is concerned, an unsatisfied required type is the sole cause for the generation of the doesNotUnderstand: message. The tool guarantees prevention of the doesNotUnderstand: message, by requiring that all required types are satisfied. The user is alerted to type mismatches, and it is the user's responsibility to satisfy them. Type mismatch usually occurs with message arguments.

We continue to morph, aware that the type mismatch was due to an intermediate morph step.

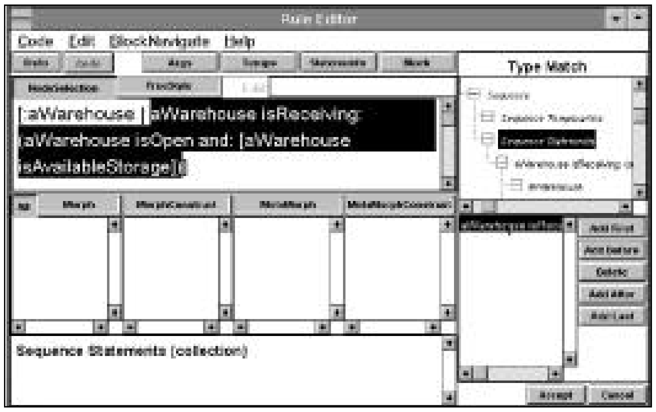
7) [:subject | subject isReceiving: **subject isNil**]
8) [:subject | subject isReceiving: **subject isOpen**]
9) [:subject | subject isReceiving: **subject isOpen isNil**]
10) [:subject | subject isReceiving: **(subject isOpen and: [nil])**]
11) [:subject | subject isReceiving: (subject isOpen and: [**nil**])]
12) [:subject | subject isReceiving: (subject isOpen and: [**subject**])] (At step 12 there is another type mismatch, again with the argument of isReceiving:.)
13) [:subject | subject isReceiving: (subject isOpen and: [**subject isNil**])]
14) [:subject | subject isReceiving: (subject isOpen and:



```
[:subject | subject isReceiving: subject]
```

VariableNode
Type: <Warehouse>
Required-type: <UndefinedObject | (kindOf: Boolean)>

Figure 4. DEMO 4.



```
[:aWarehouse | aWarehouse isReceiving:
(aWarehouse isOpen and: [aWarehouse
isAvailableStorage])]
```

Sequence Statements (collection)

Figure 5. DEMO 5.

```
[subject isAvailableStorage])]
```

We're almost done creating the action block. We now select the VariableNode block argument subject and rename it to aWarehouse by editing its name in the edit box; this renames all references in the scope of the variable.

```
15)   [:subject | subject isReceiving: (subject isOpen
and: [subject isAvailableStorage])]
16)   [:aWarehouse | aWarehouse isReceiving:
(aWarehouse isOpen and: [aWarehouse isAvailableStorage])]
```

That's it, and it took a total of 15 mouse clicks (not including list expansions) and one name edit.

Now let's say we wanted to add another statement to the action block. We start by selecting the Sequence Statements collection. A collection editor is installed in the lower right, as shown in Figure 5. We select the single statement, and then click on the Add Before button, and get:

```
[:aWarehouse |
nil.
aWarehouse isReceiving: (aWarehouse isOpen and:
[aWarehouse isAvailableStorage])]
```

The added nil statement can then be built out via morphing. The collection editor can also modify a collection of temporary-variables.

The tool has unlimited undo/redo capability, as well as the ability to alternate between NodeSelection and FreeStyle (i.e., regular) text views. Text is automatically parenthesized and formatted in the NodeSelection view (as we have seen). Also in this view, the user is able to walk up the tree by shift clicking.

Assignment to a block is not supported, nor is sending it #value. Cascaded expressions are currently not supported.

NODE WRAPPERS

The ProgramNode class hierarchy (somewhat simplified) is shown in Figure 6. Instance variables are shown with soft type (gleaned from class comments). Note that a BlockNode contains a SequenceNode, which in turn contains temporaries and statements.

```
ProgramNode (sourcePosition <Interval>)
    ParameterNode (variable <VariableNode>)
    StatementNode
        ReturnNode
        ValueNode
            AssignmentNode (variable <VariableNode>,
value <kindOf: ValueNode>)
            CascadeNode
            LeafNode
                BlockNode (arguments <Collection of:
ParameterNode>, body <SequenceNode>)
                LiteralNode (value <kindOf: Object>)
                VariableNode (name <String>)
            SequenceNode (temporaries <Collection of:
ParameterNode>, statements <Collection of: (kindOf:
StatementNode)>)
            SimpleMessageNode (receiver <kindOf:
ValueNode>, selector <Symbol>, arguments <Collection of:
(kindOf: ValueNode)>)
                MessageNode
```

The ProgramNode class-hierarchy (somewhat simplified).

A wrapper class hierarchy that parallels that of the ProgramNodes was created. The wrapper classes refer to the ProgramNode classes and extend them, but the ProgramNode classes themselves are not modified, thus keeping the compiler framework intact (Adapter pattern[1]). The wrapper hierarchy (somewhat simplified) is shown in Figure 7. (AbstractParserTraverser and MessageWrapperBlockArgument Evaluator are technically not wrappers, but are referred to as such.)

```
AbstractParserTraverser (parent, children)
    MessageWrapperBlockArgumentEvaluator
    AbstractParserWrapper (value)
        AbstractProgramNodeWrapper (type,
requiredType)
            ParameterNodeWrapper
            AbstractStatementNodeWrapper
                ReturnNodeWrapper
                AbstractValueNodeWrapper
                    AssignmentNodeWrapper
                    AbstractLeafNodeWrapper
                        LiteralNodeWrapper
            BlockNodeWrapper
            VariableNodeWrapper
         SequenceNodeWrapper
         MessageNodeWrapper
            MessageNodeWithArgumentsWrapper
   OrderedCollectionWrapper
      ParentOfUserRoot
```

The wrapper hierarchy (somewhat simplified).

Added state allows a ProgramNode to know its parent, as well as keep track of its required and current type. OrderedCollectionWrapper wraps collections pointed to by ProgramNodes; this includes SequenceNode statements and MessageNode arguments. *Manipulation* is defined as either wrapper replacement or addition/deletion of an OrderedCollectionWrapper child.

Not having the option of modifying the ProgramNode classes can be tricky. It was necessary for example, to deep-copy a ProgramNode, an ability which it lacks. The technique is to regenerate it by compiling its decompiled string.

THE USER-ROOT

The user-root *parent* hierarchy is designated as follows:

```
nil
   BlockNodeWrapper
      SequenceNodeWrapper
         ParentOfUserRoot
```

The ParentOfUserRoot is an OrderedCollectionWrapper on sequence statements and is constrained to always have one statement– the user root. Wrapper replacement is forbidden for any wrappers above the user root. Indeed, the user is aware of the user-root tree only. The user-root must have a parent because the user root needs to be replaceable, and this requires a parent wrapper. As a kind of AbstractStatementNodeWrapper, the user root has flexibility in being replaceable with wrappers of other classes.

CONSTRUCTS

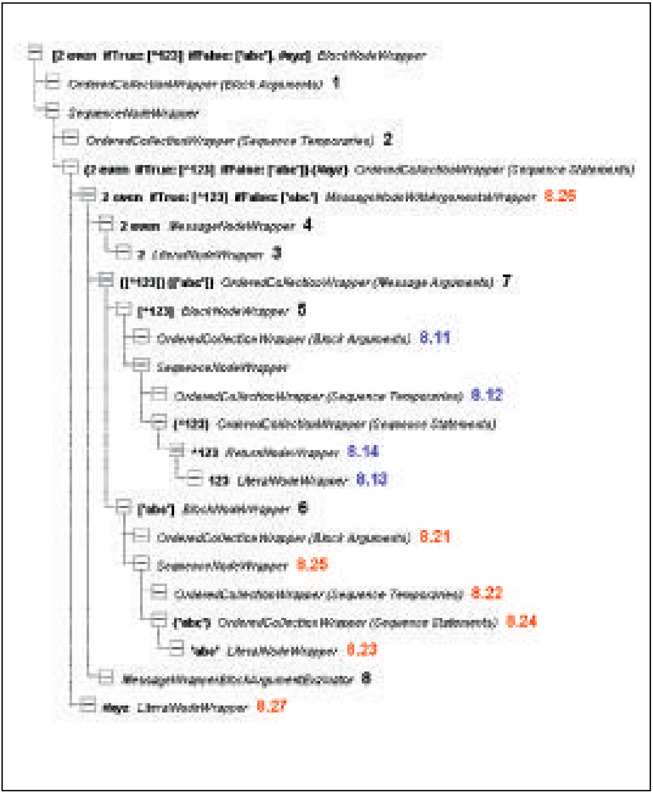
Constructs are obtained from the soft-typing information of ProgramNode instance variables (as shown in Figure 6.). For example, take a MessageNode, which is a statement. This node can be replaced with any of the subclasses of StatementNode such as ValueNode. Since the receiver of a MessageNode is itself a ValueNode, it follows that it is permissible to replace this MessageNode with its receiver, as shown in Figure 8.

Replacing a statement MessageNode with its receiver.

Construct Return message receiver (123 isNil → 123).

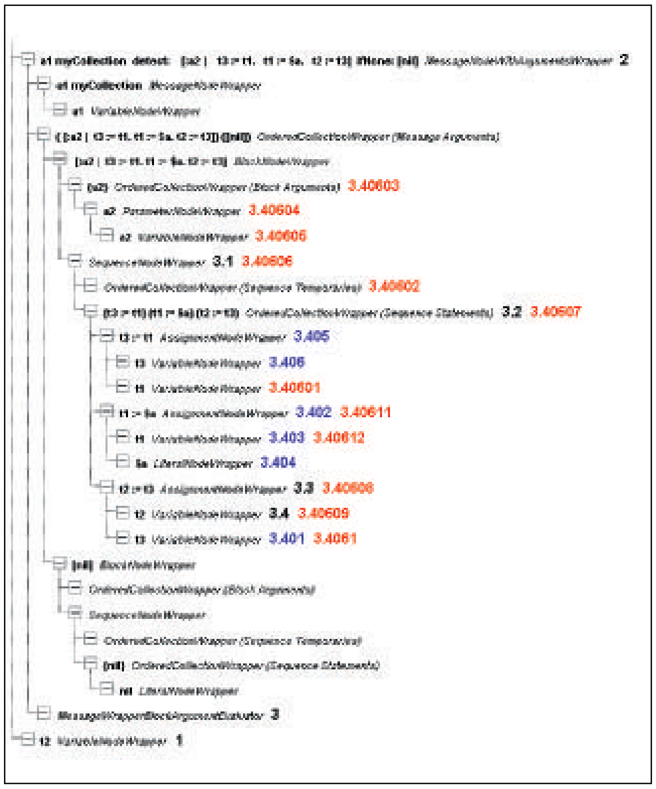
The following constructs are supported:

- Return message receiver (123 isNil → 123)
- Be message receiver (123 → 123 isNil)
- Change message selector and arguments only (123 + 456 → 123 * nil)
- Change message selector only (123 + 456 → 123 * 456)



```
[2 even ifTrue: [^123] ifFalse: ['abc']. #xyz] BlockNodeWrapper
   OrderedCollectionWrapper (Block Arguments) 1
   SequenceNodeWrapper
      OrderedCollectionWrapper (Sequence Temporaries) 2
      (2 even ifTrue: [^123] ifFalse: ['abc'])(#xyz) OrderedCollectionWrapper (Sequence Statements)
         2 even ifTrue: [^123] ifFalse: ['abc'] MessageNodeWithArgumentsWrapper 8.26
            2 even MessageNodeWrapper 4
               2 LiteralNodeWrapper 3
            ([^123])(['abc']) OrderedCollectionWrapper (Message Arguments) 7
               [^123] BlockNodeWrapper 5
                  OrderedCollectionWrapper (Block Arguments) 8.11
                  SequenceNodeWrapper
                     OrderedCollectionWrapper (Sequence Temporaries) 8.12
                     (^123) OrderedCollectionWrapper (Sequence Statements)
                        ^123 ReturnNodeWrapper 8.14
                           123 LiteralNodeWrapper 8.13
               ['abc'] BlockNodeWrapper 6
                  OrderedCollectionWrapper (Block Arguments) 8.21
                  SequenceNodeWrapper 8.25
                     OrderedCollectionWrapper (Sequence Temporaries) 8.22
                     ('abc') OrderedCollectionWrapper (Sequence Statements) 8.24
                        'abc' LiteralNodeWrapper 8.23
            MessageWrapperBlockArgumentEvaluator 8
         #xyz LiteralNodeWrapper 8.27
```

Figure 6. Branch.



```
a1 myCollection detect: [:a2 | t3 := t1. t1 := 5a. t2 := t3] ifNone: [nil] MessageNodeWithArgumentsWrapper 2
   a1 myCollection MessageNodeWrapper
      a1 VariableNodeWrapper
   ([:a2 | t3 := t1. t1 := 5a. t2 := t3])([nil]) OrderedCollectionWrapper (Message Arguments)
      [:a2 | t3 := t1. t1 := 5a. t2 := t3] BlockNodeWrapper
         (a2) OrderedCollectionWrapper (Block Arguments) 3.40603
            a2 ParameterNodeWrapper 3.40604
               a2 VariableNodeWrapper 3.40605
         SequenceNodeWrapper 3.1 3.40606
            OrderedCollectionWrapper (Sequence Temporaries) 3.40602
            (t3 := t1)(t1 := 5a)(t2 := t3) OrderedCollectionWrapper (Sequence Statements) 3.2 3.40607
               t3 := t1 AssignmentNodeWrapper 3.405
                  t3 VariableNodeWrapper 3.406
                  t1 VariableNodeWrapper 3.40601
               t1 := 5a AssignmentNodeWrapper 3.402 3.40611
                  t1 VariableNodeWrapper 3.403 3.40612
                  5a LiteralNodeWrapper 3.404
               t2 := t3 AssignmentNodeWrapper 3.3 3.40608
                  t2 VariableNodeWrapper 3.4 3.40609
                  t3 VariableNodeWrapper 3.401 3.4061
      [nil] BlockNodeWrapper
         OrderedCollectionWrapper (Block Arguments)
         SequenceNodeWrapper
            OrderedCollectionWrapper (Sequence Temporaries)
            (nil) OrderedCollectionWrapper (Sequence Statements)
               nil LiteralNodeWrapper
   MessageWrapperBlockArgumentEvaluator 3
t2 VariableNodeWrapper 1
```

Figure 7. Loop.

- Return block's first and only statement ([123] → 123)
- Enclose statement in block (123 → [123])
- Return assignment value (t1 := 123 → 123)
- Be assignment value (123 → t1 := 123)

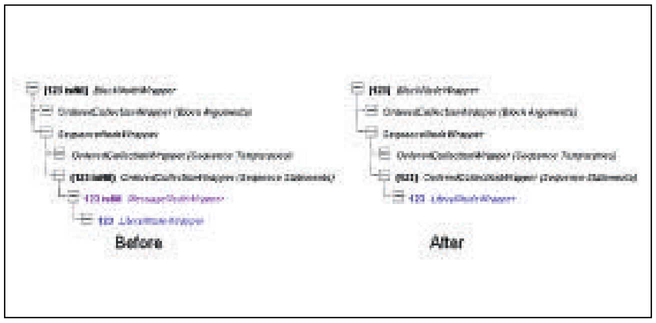


Figure 8. Construction.

- Remove up-arrowT> (^123 → 123)
- Add up-arrow (123 → ^123)

MANIPULATION VALIDATION

Each potential manipulation must be validated to be made available to the user. A manipulation is invalid if it hasn't been implemented for example, like the replacement of a message receiver. A manipulation is invalid if it is syntactically incorrect. For example, the definition of a temporary variable may not be deleted if the variable is currently referenced.

We also check the ramifications for message receivers if the type of a temporary variable were to change. For a message-receiver whose type is determined by a temporary, we ensure that the receiver's required type is satisfied. For example, t1 isEmpty, where the receiver's required-type is <kindOf: Collection>. If the type of t1 were to change from <ByteString> to <SmallInteger>, for example, the check would fail, because <kindOf: Collection> would not be satisfied with <SmallInteger>. This check is accomplished by simulating the manipulation on a parallel test tree to preview the results.

CORRECT SOURCE POSITION

Wrapper source-position information enables the user to select the wrapper by clicking on it in a text view. A pristine ProgramNode tree will have correct source-position information stored in its nodes. Once the tree is manipulated, however, this information is no longer guaranteed to be in synch with the decompiled string. The technique is to create a parallel ProgramNode tree after every manipulation, by compiling the decompiled string of the modified wrapper tree. A wrapper seeks its correct source position from its parallel node counterpart.

Instead of directly replacing a node in the tree, why not do so indirectly by replacing the corresponding string component in the overall string? For a BlockNode, for example, replace the string '[nil]' in the correct spot in the overall string. Then, from this overall string create a fresh wrapper tree that would then have the correct source position. This approach, however, also has a drawback; The state information in the pre-manipulation wrapper tree must always be transferred to the post-manipulation wrapper tree.

In the next article, we plan to conclude by looking at typing, traversal of the wrapper tree, treatment of blocks, and creation of the wrapper-tree. S

Reference

1. Gamma, E. *et al. Design Patterns:* ELEMENTS of REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, Reading, MA, 1995.

Paul Davidowitz is a senior developer at Andersen Consulting. He can be reached at paul.davidowitz@ac.com.