Taking Exception to Smalltalk

Bob Hinkle* and Ralph E. Johnson

University of Illinois at Urbana-Champaign

Part 2

Last month we described the system-independent parts of our implementation of an exception-handling system. This month's article describes those parts of our implementation that rely on Smalltalk V/286 specifics, with special emphasis on contexts. As a result, this article should interest anyone needing to do low-level hacking in V/286, since it describes the architecture of process stacks, how contexts fit onto that stack, and how temporaries are layed out in contexts. The ability to examine and manipulate contexts is both powerful and useful, and it's been exploited in two non-exception-handling efforts that we know about. The first and foremost is the system debugger, which uses all of contexts' capabilities, including the modification of local variables and the resumption of execution at any point in the stack. The other example is a backtracking system for Smalltalk developed by Wilf LaLonde and Mark Van Gulik [1], which, like our exception handler, uses contexts to implement non-standard control flow.

The last few pieces to our implementation are the system-specific methods in class `Exception`, extensions to the fundamental classes `Process`, `Context`, and `HomeContext`, and the addition of three new context-related classes. The changes in these classes comprise extensions to Digitalk's base that make processes and particularly contexts easier to work with. The same changes are not necessary in ParcPlace's Smalltalk-80, which provides all the functionality we need and more.

* Supported by a fellowship from the Fannie and John Hertz Foundation.

The Machine-dependent Implementation

There are three Exception methods we still need to describe: fetchHandlerBlock:, restart, and return. Each of these methods depends on some specific aspects of V/286.

We begin with fetchHandlerBlock:, the method used by propagatePrivateFrom: to find the correct handler for the receiving exception. FetchHandlerBlock: is implemented as:

```
fetchHandlerBlock:   startContext
    startContext sendersDo: [ :ctxt |
        (ctxt selector == #handle:do:
            and: [ctxt receiver accepts: signal])
            ifTrue: [handlerContext := ctxt.
                ^ctxt at: 4]].
    ^nil
```

In general, startContext will be the value of the exception's signalContext instance variable, which is the context of the raise message. The message sendersDo: is used to iterate down the context stack from startContext, applying the block to each context in turn. The block checks each context looking for a handler for the exception; the correct handler context will be the first one reached where handle:do: was sent to the exception's signal or one of its parents (which is what the accepts: method checks for). When such a context is found, it's remembered as the handlerContext, and the object in its fourth slot is returned. This object will be the block that was passed as the first parameter of the handle:do: message. It is fourth because of the order Digitalk stores local variables in contexts, with first slots for block arguments in reverse order of their appearance, followed by temporaries in reverse order, then parameters in reverse order. Figuring this out requires some knowledge of the context layout in V/286; we'll describe that in more detail when we discuss contexts below.

The return method is implemented in terms of returnDoing:, which itself is implemented as follows:

```
returnDoing:   aBlock
    "The stack is unwound to the context of the handle:do:
    message that caught this Exception, at which point
    aBlock is evaluated and its value returned as the value
    of the handle:do: message."

    | answer |
```

```
        answer := aBlock value.
        self handlerContext unwindLaterContexts.
        (self handlerContext at: 2) value: answer
```

This is analogous to the implementation for proceedDoing:--the only difference is in accessing the block that will (when evaluated) return into the right context. In this case returnBlock is stored in the handler's context (recall our definition of handle:do: from Part 1) and is accessible in the second slot of the context's array of temporaries. So evaluating the returnBlock returns from the handle:do: context as desired. As with proceedDoing:, though, the method must call unwindLaterContexts first to make sure unwind blocks are evaluated.

Implementing restart relies on restartAt:, a Digitalk-provided method for class Process, as seen in the following code:

```
    restart
        "Restart the #handle:do: context."

        | index process |
        handlerContext unwindLaterContexts.
        process := handlerContext process.
        index := process frameIndexOf: handlerContext.
        process restartAt: index
```

This method makes a process restart execution at an arbitrary context in its context stack. First, as before, the exception unwinds all contexts above its handlerContext. The exception then finds the handlerContext's index in its process, and tell its process to restart execution there.

To motivate the changes to the Process and Context classes, we need first to describe how these classes relate in the base system. Process in V/286 is a subclass of OrderedCollection; its indexed instance variables are used to store information about the stack of unresolved message sends. Conceptually we think of each message send as being represented by a Context object. However, for optimization purposes, V/286 only creates HomeContexts for certain method invocations. (In particular, they create a HomeContext only if the method that's evaluated contains a block.) This dual representation is potentially troublesome, so we hide it behind two new context-related classes. Before looking at these classes, though, we need to understand the layout of Process' stack. Each message send receives a stack frame of five or more slots on the stack, with the following layout:

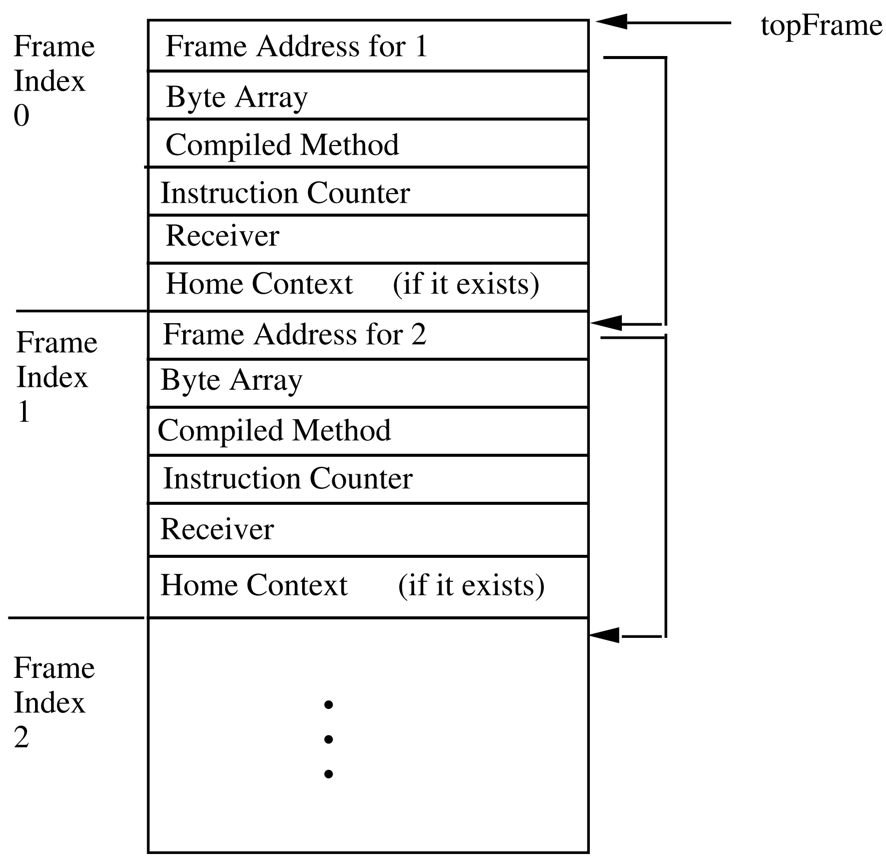

Figure 2: Process stack layout

Each stack frame begins with the frame address of the previous message. This frame address is unique for each message send, and it persists as long as the message is on the stack. In addition to this address, message sends can be referenced by their frame index, which is the message send's position on the stack. The topmost (i.e., most recent) send is at frame index 0, the previous send is at index 1, and so on.

After the frame address comes the byte array (which is Smalltalk's compiled representation of the method), the compiled method, the instruction counter, and the message's receiver. If a HomeContext exists for the frame, it will be stored in the sixth slot. If there is no HomeContext for the frame then there will be slots for each of the parameters and temporary variables (in the same order referred to above: block arguments in reverse order followed by temporaries in reverse order followed by parameters in reverse order).

The Process class provided by Digitalk comes with a method, contextFor:, that returns the context for a given frame index. This has three problems for our purposes: first, the context may not exist, in which case nil is returned; second, the context returned for blocks (which are instances of class Context in V/286) is their HomeContext, which is not the same as their frame on the context stack; and third, since the frame index changes as execution proceeds, we really need to use the frame address to identify our contexts. So we added two methods to Process, indexOfAddress: and groundedContextFor:. The method indexOfAddress:, which converts a frame address into a frame index, first checks if the input frame address matches the address of its top frame. If it is, the receiving process returns 0 as the associated frame index. Otherwise the process returns the index of the first frame whose frame address matches the input.

```
indexOfAddress:   frameAddress
    | index address |

    frameAddress = topFrame
        ifTrue: [^0].

    index := 1.
    [address := self frameAt: index - 1 offset: 0.
    frameAddress = address]
        whileFalse: [
            address = 0
                ifTrue: [^nil].
            index := index + 1.
        ].

    ^index
```

The other addition to Process is a method called groundedContextFor: that is an extended version of contextFor:. It differs from contextFor: only in that it always returns a context-like object, whether or not a real HomeContext exists for the frame requested:

```
groundedContextFor:   frameIndex
    | frameAddress |

    (self methodAt: frameIndex) hasBlock
        ifTrue: [
            (self homeFrameOf: frameIndex) = frameIndex
                ifTrue: [^GroundedContext
                        forIndex: frameIndex
                        forProcess: self
                ]].
    ^PseudoContext
        forIndex: frameIndex
        forProcess: self
```

This method makes use of the two new classes we added. A GroundedContext object is returned when a real HomeContext is available for the frame. (We return a GroundedContext rather than the HomeContext itself because GroundedContexts have additional behavior--in particular they know the process they belong to.) When a real HomeContext is not available for the frame, or when the frame corresponds to a block's activation, the method creates a PseudoContext object. This object knows only the frame address of the frame it represents--but using the frame address it can behave exactly like a normal context. Thus, PseudoContexts and GroundedContexts seem identical externally, hiding the difference between frames with HomeContexts and those without.

PseudoContext and GroundedContext are designed to fulfill the same interface, so we also created an abstract class called AbstractContext that is a common superclass for the two. AbstractContext defines the interface, and it also implements a number of methods by depending on a few methods from its subclasses. In particular, AbstractContext defines the method unwindLaterContexts as:

```
unwindLaterContexts
    "Search down the stack, starting with the current
    context, evaluating the unwindBlock in every
    Context>>valueOnUnwindDo: or
    Context>>valueNowOrOnUnwindDo: context. Stop at
    the receiver."

    | s |
    self thisContext
        sendersDo:
            [:ctxt |
            ctxt == self ifTrue: [^self].
            (ctxt receiver isKindOf: Context)
                ifTrue: [(s := ctxt selector) ==
                        #valueOnUnwindDo:
                    ifTrue: [(ctxt at: 1) value]
                    ifFalse: [s == #valueNowOrOnUnwindDo:
                        ifTrue: [(ctxt at: 2) value]]]]
```

This is similar to the fetchHandlerBlock: method. It looks down the message stack starting at the current context, looking for any context for the valueOnUnwindDo: or valueNowOrOnUnwindDo: messages. If it finds one, it evaluates the unwind block, which is available in the first slot of the valueOnUnwindDo: context or the second slot of the valueNowOrUnwindDo: context.

In addition, AbstractContext defines the sender method as follows:

```
sender
    ^process groundedContextFor:
        (process indexOfAddress: self address) + 1
```

AbstractContext defines the at: and at:put: methods to provide access to its underlying context's array of temporary variables. The at: method is implemented as:

```
at:  anInteger
    | index |
    index := process indexOfAddress: self address.
    ^process tempAt: index number: anInteger
```

At:put: works the same way except that it stores into the slot (using tempAt:number:put:) rather than reading from it. The definition of the address method used in sender and at: differs for PseudoContext and GroundedContext, with the former returning the value of its frameAddress instance variable and the latter returning the frameOffset of its underlying HomeContext.

Finally, there is one method, thisContext, we added to Object. Like sender, this is a feature that's built in to Smalltalk-80. Unlike the sender method, though, thisContext is supported as a pseudo-variable (like self) in Smalltalk-80, but we implement it as a method for V/286:

```
thisContext
    ^[] homeContext sender
```

This completes the implementation of our exception-handling system. After adding this package to your V/286 system, you can introduce the use of signals to identify common or important errors, and so support dynamic responses to errors in future work. Besides this practical benefit, our addition of signal handling to V/286 is important as an illustration. Because processes and contexts are programmer-manipulable objects, we were able to extend the functionality of the low-level system to support our needs as application programmers. In particular, methods for Exception needed to reflect on the system's operation by knowing about and accessing V/286's representation of contexts and processes. Without that ability, we'd have been unable to make the necessary changes--only language implementers could make them, by changing the language and the compiler themselves.

It’s also interesting to compare our implementation with ParcPlace’s system. While we’ve provided much of the same functionality, Smalltalk-80’s exception handling has two advantages over ours. First, their system is more efficient than ours because it is supported by the virtual machine instead of being implemented entirely in Smalltalk. They have several important optimizations to speed up expensive operations such as traversing the context stack looking for the next exception handler or unwind block. In addition, the reflective nature of our implementation is slower because we rely on more layers of message sends and abstractions--this is a problem that will exist until reflective programming can be recognized and optimized by the compiler. Smalltalk-80 also behaves better because, as we noted last month, normal method returns are treated just like returns from exceptions, so that unwind blocks will be executed if skipped over either in exception handling or in normal computation. We couldn’t provide the same function because the semantics of method returns are hard-wired into the V/286 virtual machine; we would have benefited from a more reflective implementation in which method returns could be modified by the programmer. In the future we hope to write other articles that highlight reflective aspects of Smalltalk, and to see other practical benefits of objectifying system internals.

References

[1] Wilf R. LaLonde and Mark Van Gulik. Building a Backtracking Facility in Smalltalk Without Kernel Support. In *Proceedings of OOPSLA ‘88, Object-Oriented Programming Systems, Languages, and Applications*, pp. 105 - 122, Novermber 1988. Printed as SIGPLAN Notices, Volume 23, Number 11.