A Remote Debugger For Modtalk

A Senior Project by Josh Fridstrom

Northern Michigan University, April 2015

Contents

About Modtalk

My senior project is to write a debugger for Modtalk. Modtalk is a compiler tool-chain for the Smalltalk programming language which adds support for compile time namespaces. Modtalk compiles Smalltalk source code, stored in modules, to an in-memory model. A builder tool runs across the model generating executable code in various forms. Prior to my senior project, we supported two target runtimes. The first of these is a threaded interpreter runtime where each interpreted operation (op) correlates to about one abstract machine code instruction. The second supported runtime is native machine code. That's right, we can compile directly to native x64 machine code.

Code Organization

The code is organized into three separate code domains. The first of these is the Modtalk virtual machine. The vm is written in C and lives in a git repository. The virtual machine is where the different runtimes are written that execute compiled Modtalk code. Additionally, the vm handles various primitives that Modtalk programs can use. These include, but are not limited to, small integer arithmetic, basic array operations, and IO. I worked in this code domain quite a bit by adding a new runtime, adding several primitives for various features, and writing a bunch of code that handles socket IO for the remote debugger. The second code domain is Modtalk code. This is Smalltalk code that is stored into modules that are compiled by the Modtalk compiler tools and built into a mixture of C code and assembly code that can be made into a single executable. I wrote much of the debugger code in the context of a Modtalk Subsystem which is akin to a Java package or a Python module. The Debugger Subsystem contains definitions for a Debugger Class as well as many other classes used by the Debugger objects. Instances of the Debugger Class control the execution of the process being debugged. The Debugger Subsystem is meant to be included with a program that you wish to debug. The existence of the Debugger Subsystem will allow the user to take advantage of the Debugging features. The last code domain is Pharo Smalltalk. The Modtalk IDE and compiler tools are all written in Pharo Smalltalk. I wrote the remote debugger client in Pharo Smalltalk so that I could integrate it with the IDE and use the program model built by the compiler tools. My remote debugger is written in such a way that you could write a program in any language (so long as you can open a socket) and debug a running Modtalk program.

The Bytecode Interpreter

The first step to creating the debugger was to add an additional target runtime, a bytecode interpreted runtime. I needed a bytecode runtime to make it easy to set breakpoints. More on that later. I worked in two separate development domains to make this work. First, I needed to extend the target builder that runs across the program model to generate the appropriate bytecode sequences to be executed. This was done in Pharo Smalltalk. Second, I needed to write the interpreter to execute the bytecode sequences. This was done in the Modtalk vm. Developing the bytecode interpreter was pretty straightforward. Having a threaded interpreter made it easy to get going. Each byte corresponds to a single op or a grouping of related ops and the arguments for the ops were encoded in the following byte or two. So, I simply mapped bytes or pairs of bytes to threaded ops which we already had debugged and working.

Command-Line Debugger

Once I had a bytecode interpreter, I was able to get started on my debugger. When a program is run with the Debugger Subsystem, the Debugger Subsystem immediately takes over and goes into a print/eval loop. This allows users to enter commands via a command line interface. At this point users can set breakpoints run the program. When the run command is issued, the Debugger Subsystem lets the program continue executing as normal. When a breakpoint is hit by the running program the current running process is suspended, an instance of the Debugger class is created and a separate debugger process is created and scheduled. In this new debugger process, the Debugger object goes into another print/eval loop where more commands can be given by the user. Commands that can be issued in this state include continuing program execution, requesting context information such as temporary variables or methods arguments, and dumping the stack.

Setting breakpoints is one of the most important features that I needed to have working for my debugger. Users can set breakpoints on message sends within a method. Every message send corresponds to a send bytecode in the compiled bytecode sequence for that method. Setting a breakpoint then becomes a simple task of replacing the send bytecode with a special breakpoint bytecode. The debugger keeps track of which methods have breakpoints and what offset into the bytecode sequence the breakpoint is set. When the program is executing, instead of reading the send byte and performing a send operation, the breakpoint byte is read and a special breakpoint operation is performed. The breakpoint operation is what does the suspension of the current running process and creates the new debugger process as was mentioned earlier. When a continue command is entered by the user, the debugger process terminates and the previously running process can be scheduled to again.

Command	**Arguments**	**Desciption**
break set	Subsystem.ClassName methodName byteIndex	sets a breakpoint at byteIndex in a method
run		
continue		
into		
over		
dump	[n]	show stack trace back n frames or back to start
arg stack	frameIndex index	returns argument found index off the frame
arg env	frameIndex envIndex index	returns argument found index into the envIndex env at the frameIndex frame
temp stack	frameIndex index	returns temporary found index off the frame
temp env	frameIndex envIndex index	returns temporary found index into the envIndex env at the frameIndex frame

Figure 1: CLI Commands

Figure 1 is a table describing the commands available to the user to control program execution, set breakpoints, and access arguments and temporaries. Allow me to explain them. In order to set a breakpoint, the debugger needs to know what method to set the breakpoint in as well as where in the method to do so. The Subsystem.ClassName argument specifies a fully qualified class name. Modtalk programs are composed of Subsystems, in which classes are defined. Class names must be unique within a subsystem, so the subsystem name and the class name together will correspond to no more than one class. Once the class is found, the method can be looked up by its name. The final argument, the byteIndex, specifies how far into the compiled bytecode sequence to place the breakpoint as explained in the previous paragraph. “Run” and “continue” merely terminate the debugger process and let the program continue running. “Into” performs the send that is currently halting the program and stops at the next available send. “Over” performs the send and halts at the next available send within the method currently breaking in. The “dump” command prints a stack trace. The next commands are used to access temporary variables and arguments to methods. The reason there are four commands instead of two is because Modtalk supports full closures. We would like most arguments and temporaries to be stack allocated because stack allocations are fast. However, sometimes a closure is created that will outlive the current stack activation. Variables referenced by this closure must be allocated in the heap. When a closure is created, it has a heap allocated environment where temporaries and arguments are allocated. Because some argument and temporary variables are stores on the stack and others are stored in heap allocated environments, and there is no way to tell the different once a program has been built to executable code, four different commands are required to access them. Sometimes closures are nested inside other closures and heap allocated environments are chained together. The envIndex argument says which environment in the chain to look in for the argument or temporary.

The Debugger Subsystem includes a Debugger class which describes the behavior of debugger objects. A debugger needs to know about the process it is currently debugging. Modtalk processes have different representations. The state of the registers for each process is stored in C memory. Prior to my project, Modtalk process objects could be created and the state and various other information about the process could be accessed and changed; however, the registers lived in C memory and could not be directly accessed by a Modtalk program without going into a C-primitive. I needed to be able to access the registers so that I could unwind the stack, access arguments and temporaries, as well as execute code on the other process. Instead of creating primitives for each different action that I needed to perform on the registers, I decided to objectify them so that I could manipulate them from a running Modtalk program. This turned out to be really useful as it gave me a simple way to access all the data that I needed as well as control the execution of the process being debugged.

Once I had a mostly usable command line debugger, I realized how agonizing the experience of using it was. You have to know a lot about how Modtalk activation frames are laid out in order to use the debugger and you have to somehow determine what variables are stored on the stack and which ones are allocated in the heap. Originally, I had planned on writing a decompiler to show the user the contents of methods so that they could determine which arguments and temporaries were stack allocated and which ones were heap allocated, but then I realized that it would be much easier if I had the program model that the executable code is generated from. The program model contains information like the names of variables and whether those arguments live on the stack or in the heap. Instead of writing a decompiler so that the user could figure out that information, I decided to jump right to the remote debugger so that I could use the program model to calculate the information for the user.

Remote Debugger

The remote debugger still has the debugger subsystem as a part of the program under development just like the command line debugger. The difference is that when the debugger process is ready to accept the next command, it will now read the command from a socket instead of from standard input. I spent several days designing the state machine for the remote debugger. It has gone through several changes since then, but once it was ready to implement I had once problem. We didn't have sockets in Modtalk yet. Instead of writing fully objectified sockets and having to worry about host IO signals and all that, I decided to do the easier thing. When a program is launched for debugging, a pthread is started which creates a socket and waits for a debugger client to connect. Meanwhile the Modtalk program continues execution as normal and just like before, the Debugger Subsystem takes control and waits for the next command. This time instead of reading from standard input, I have an inbox queue that the Debugger Subsystem will wait for something to appear in. The socket thread, in the spawned pthread, will read serialized commands from the socket, parse them, and store the program commands into the queue. Similarly, when the Debugger Subsystem performs the command, it places the reply into an outbox queue where the socket thread will read a command, serialize it, and send it over the socket. Surprisingly little code needed to be changed in the Debugger Subsystem to make this pretty significant change. This allowed me to write a Debugger client in Pharo Smalltalk that has a copy of the program model. I am also able to integrate the Debugger client with the Modtalk IDE so that setting breakpoints on a method is as simple as it is in Eclipse, Xcode, or Visual Studio.

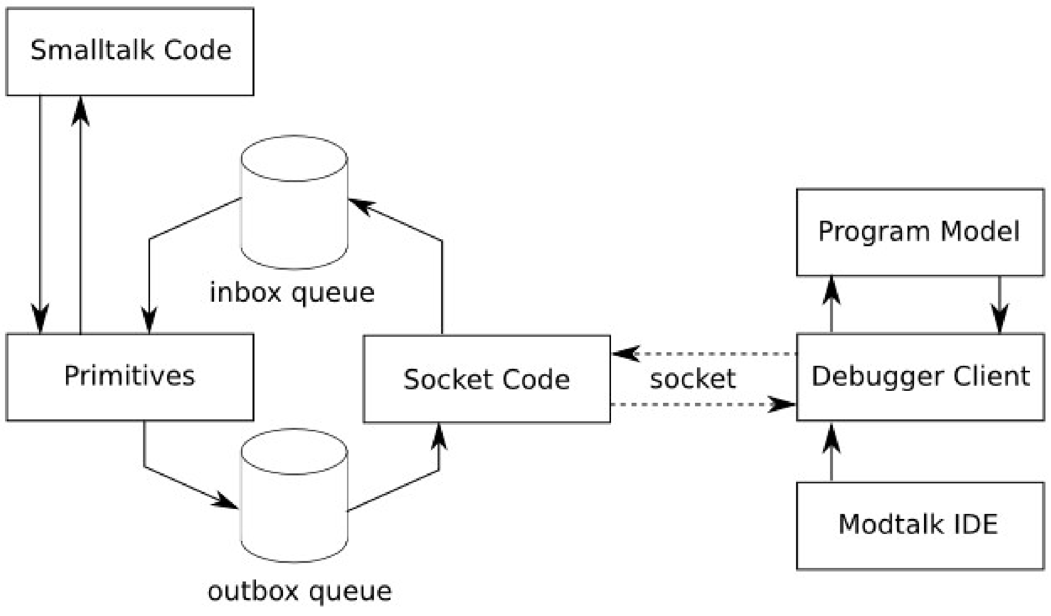

Figure 2: Remote Debugger

What I Learned

This senior project has been a fantastic learning experience for me. One of the most important lessons that I learned was how painful it is writing a debugger when I don't have a debugger to debug it. Some of the biggest challenges that I have had to overcome have to do with dealing with multi-threaded and multi-process programs. Figuring out how to handle appropriate synchronization and avoiding deadlocks was very neat. I also learned a lot about the value of a well specified state machine. I had to make several changes to my state machine during the course of the remote debugger development, but each change was easy to make and almost always done properly on the first try because I always updated my state machine documentation prior to making the change which validated the design. Figure 3 shows the current state of the remote debugger state machine.

	States	Connection Pending	Options Negotiation	Connected	Disconnected
Commands					
Connection Request	0	handleConnectionRequest	handleError	handleError	handleError
Options Negotiation Initiation	1	handleError	handleError	handleError	handleError
Options Negotiation Confirmation	2	handleError	handleOptionsNegotiationConfirmation	handleError	handleError
Options Negotiation Cancellation	3	handleError	handleOptionsNegotiationCancellation	handleError	handleError
Options Negotiation Ack	4	handleError	handleError	handleError	handleError
run	5	handleError	handleError	handleRun	handleError
break set	6	handleError	handleError	handleBreakSet	handleError
continue	7	handleError	handleError	handleContinue	handleError
into	8	handleError	handleError	handleInto	handleError
over	9	handleError	handleError	handleOver	handleError
dump	10	handleError	handleError	handleDump	handleError
arg stack	11	handleError	handleError	handleArgStack	handleError
arg env	12	handleError	handleError	handleArgEnv	handleError
temp stack	13	handleError	handleError	handleTempStack	handleError
temp env	14	handleError	handleError	handleTempEnv	handleError
Break Set Success	15	handleError	handleError	handleBreakSetSuccess	handleError
Break Set Failure	16	handleError	handleError	handleBreakSetFailure	handleError
Dump Stack Reply	17	handleError	handleError	handleDumpReply	handleError
Object Reply	18	handleError	handleError	handleObjectReply	handleError
Error Reply	19	handleError	handleError	handleMTError	handleError
Quit	20	handleError	handleError	handleQuit	handleError
Total States:	**21**				

Figure 3: Remote Debugger Protocol State Machine

Technologies and Libraries Used

I used git for a number of different parts of the project, including the vm work, the Pharo Smalltalk work, as well as the Debugger Subsystem work. I used the pthread library to get cross-platform thread support for the socket thread for the remote debugger. I used the readline library to get user input from the command line because gets gave me compiler warnings (seriously, I used gets for a while because I thought it was going to be temporary, but then switched to readline when I realized that a command line debugger is not a bad thing to have around). I used Pharo Smalltalk for development of the target builder (for bytecode runtime support) as well as the Debugger Client that integrates with the Modtalk IDE. I used gdb for my C debugging needs.

Grading

Bytecode interpreter	14/14
Evaluation control protocol design	10/10
Evaluation control implementation between processes w/shared queues	28/33
Debugger control interface design	10/10
Decompiler	00/10
Debugger control command line interface	17/23

Extra Credit

Remote object framework	+05
Evaluation control protocol implementation #2	+05
Evaluation control protocol implementation #3	+03
Debugger control interface implementation #2	+10
Total	79/100
	+23
	102/100

Grade: A