Multi-Level Virtual Machine Debugging using the Java Platform Debugger Architecture[⋆]

Thomas Würthinger[1], Michael L. Van De Vanter[2], and Doug Simon[2]

[1] Institute for System Software
Johannes Kepler University Linz
Linz, Austria
[2] Sun Microsystems Laboratories
Menlo Park, California, USA
wuerthinger@ssw.jku.at, michael.vandevanter@sun.com, doug.simon@sun.com

Abstract. Debugging virtual machines (VMs) presents unique challenges, especially *meta-circular* VMs, which are written in the same language they implement. Making sense of runtime state for such VMs requires insight and interaction at multiple levels of abstraction simultaneously. For example, debugging a Java VM written in Java requires understanding execution state at the source code, bytecode and machine code levels. However, the standard debugging interface for Java, which has a platform-independent execution model, is itself platform-independent. By definition, such an interface provides no access to platform-specific details such as machine code state, stack and register values. Debuggers for low-level languages such as C and C++, on the other hand, have direct access only to low-level information from which they must synthesize higher-level views of execution state. An ideal debugger for a meta-circular VM would be a hybrid: one that uses standard platform-independent debugger interfaces but which also interacts with the execution environment in terms of low-level, platform-dependent state.
This paper presents such a hybrid architecture for the meta-circular Maxine VM. This architecture adopts unchanged a standard debugging interface, the Java Platform Debugger Architecture (JPDA), in combination with the highly extensible NetBeans Integrated Development Environment. Using an extension point within the interface, additional machine-level information can be exchanged between a specialized server associated with the VM and plug-in extensions within NetBeans.

1 Introduction

Higher level programming languages are increasingly implemented by a *virtual machine* (VM), which is implemented in a lower-level language, which is in turn compiled into the machine language of each target platform. Standard debuggers for the VM *implementation language*, often C or C++, suffice in simple situations, but not when parts of the VM (for example critical libraries) are written

[⋆] This work was supported by Sun Microsystems, Inc.

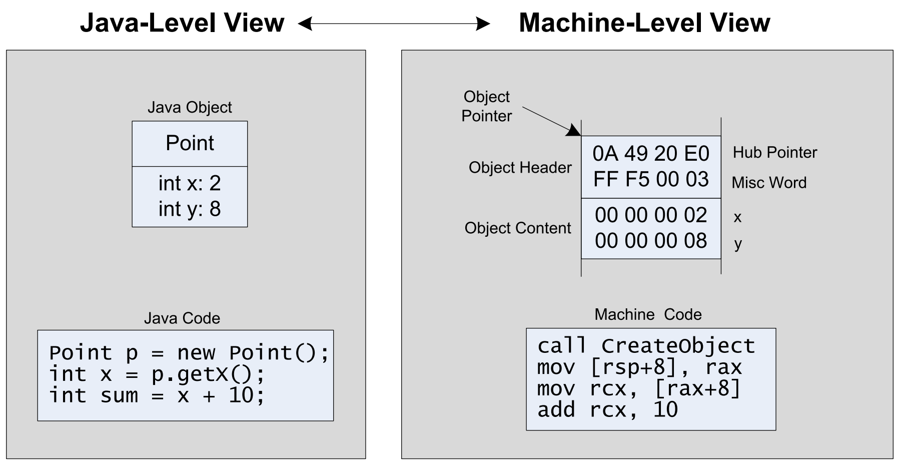
in the *implemented language*. This is also the case for applications written in the Java^{TM}programming language [4] that combine Java and C code via the Java Native Interface(JNI) [5]. This creates a demand for *mixed-mode debuggers* that support both languages: implementation and implemented.

The debugging challenge is even more complex and subtle for meta-circular VM implementations where the implementation and implemented languages are *one and the same*. A solution requires what we call *multi-level debugging*. A VM developer would like to debug at the source-level of the implementation language, but since that language is also being implemented by the VM, one must often drop to a lower level: the *machine language* of the platform into which the implementation is compiled. At this lower level one must be able to examine every aspect of machine state (registers, stacks, data layouts, compiled code, etc.) and to interpret that state in terms of the implementation language, even when the implementation may be broken. Such a tool is necessarily specialized, with the consequence that the many advantages of debugging with a modern Integrated Development Environment (IDE) are unavailable.

For example, the *Maxine Inspector* is a multi-level debugger that is of necessity highly specialized for the meta-circular *Maxine Virtual Machine* [6], for which Java is both the implementation and implemented language. The Maxine Inspector is an out-of-process, machine-level debugger that has the additional ability to interpret machine data at multiple levels of abstraction; it does this through extensive code sharing with and knowledge of the Maxine VM implementation. For example, Java objects can be viewed either abstractly or in terms of a concrete memory layout that may vary across implementation platforms and VM configurations. Java methods can be viewed as source code, bytecodes, or machine code produced by one of Maxine's compilers. Register values and memory words can be interpreted either as bits, as primitive values, as addresses, or as pointers to known Java objects. Figure 1 shows a Java object and a Java method, each viewed in both in source- and machine-level representations.

There are Java debuggers as well as machine code debuggers, but to the best of our knowledge, no system successfully combines both worlds as does the Maxine Inspector. The original Inspector, however, stood alone and lacked both the productivity features and sophisticated user interface that Java programmers expect. As an alternative to replicating those advantages, we explored integrating the core of the Inspector, the out-of-process *Inspector Debugging Agent* that reads and interprets machine state, with the extensible *NetBeans IDE* [14].

This has been made possible through development of a new framework for integration that depends on NetBeans support for the *Java Platform Debugger Architecture* (JPDA) [13], which specifies contracts between a Java VM and a Java debugger. This framework depends on JPDA's *Java Debug Interface* (JDI) [11] and uses JPDA's *Java Debug Wire Protocol* (JDWP) [12] to communicate with the debugged process over a network stream. This approach emphasizes Java-level functionality and is extended when needed for displaying additional machine-level information. A new "protocol within the JDWP protocol" enables



```
Point p = new Point();
int x = p.getX();
int sum = x + 10;
```

```
call CreateObject
mov [rsp+8], rax
mov rcx, [rax+8]
add rcx, 10
```

Fig. 1. Multiple view levels on a Java object and Java code during debugging.

the transfer of extended kinds of information via JDWP, a technique of general interest to VM developers seeking such extensions to debugging support.

Section 2 introduces this technique in the context of the Maxine VM and Inspector. Section 3 describes the extended JDWP server implementation added to the Inspector, along with a new mechanism using Java dynamic proxy classes [10] to transmit additional information via the unchanged JDWP protocol. Section 4 shows how a new "protocol with the JDWP protocol" communicates with NetBeans, in particular with plug-in extensions that allow the debugger to use the mechanism. Section 5 comments on the advantages of this approach, Section 6 reviews related work, and Section 7 concludes.

2 System Architecture

The Maxine Inspector requires almost no active support from the Maxine VM; this is necessary because there is no separate implementation language whose own implementation can be relied upon. The Inspector runs in its own process, reads VM memory via inter-process communication, and implements basic debugging commands by managing the VM process. The original Inspector internally comprises two software layers: an *agent* that both communicates with the VM and interprets its state, and a *front end* that adds graphical views and user interaction.

The alternative architecture developed for multi-level Maxine debugging with NetBeans uses JDWP: an asynchronous protocol that defines a standard for communication between Java debugger and VM. Important JDWP command groups are:

- *VirtualMachine*: get general information, loaded classes, current threads.
- *ReferenceType*: reflective information about types, class fields and methods.

– *ObjectReference*: retrieve the values of an object; invoke an instance method.
– *ThreadReference*: get data about the current call stack and thread status.
– *EventRequest*: install callbacks for events, e.g. class loading and breakpoints.

The Java Platform Debugger Architecture specifies how a Java debugger, using a standard interface (JDI) and wire protocol (JDWP), can connect to a remote server for debugging Java programs via a network connection.

Figure 2 shows how Maxine system components interact in this architecture; new components are dark gray, and the others are unchanged. The new Maxine JDWP Server delegates commands to the existing Maxine Inspector Agent. New plugins extend the NetBeans debugger and communicate directly with the JDWP server. These plugins use both the standard JDWP protocol and a new technique, described in Section 4, to access additional information not directly supported by JDWP. Examples of such information include the address of objects in memory and compiled machine code for a Java method.

3 The Maxine JDWP Server

Most JDWP commands are *queries* from the client (debugger) that produce answers from the server (agent), for example to gather information about loaded classes and thread state. Events that originate in the VM are transmitted to the client only when the client has *registered* for the events with a JDWP command. Current Maxine VM limitations delayed a complete implementation of the JDWP protocol by the server, but the implemented subset already suffices for debugging the Maxine VM using NetBeans.

In addition to standard Java debugging operations, the Maxine JDWP Server can set machine code breakpoints, perform a machine code single-step, examine memory contents, and more. When a breakpoint is reached, the server transmits information about threads and their instruction pointers. The NetBeans debugger always needs a correct Java bytecode position, which is then matched back to the source code and highlighted as the current line in the program. The server calculates the bytecode position based on the current machine code address; the position can be either exact or approximate, depending on which Maxine compiler produced the code. For setting a breakpoint in the Java source code, the server performs the reverse approximation, because the command sent by the client Java debugger contains the bytecode location only.

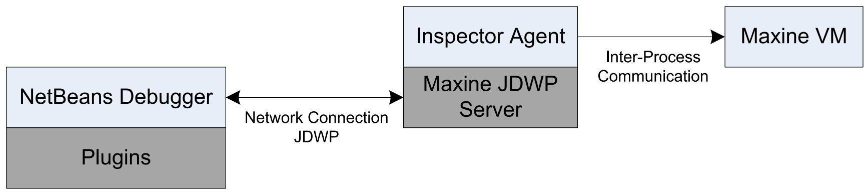
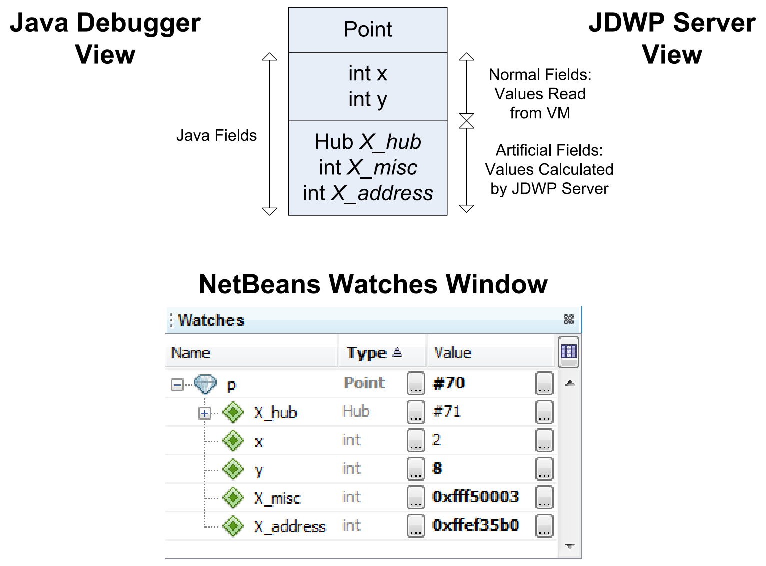


Fig. 2. System architecture.

The Maxine JDWP Server holds information about all loaded classes. Requests for object field access are delegated to the Maxine Inspector Agent, which reads the raw bytes and converts them, if possible, to valid JDWP types. This allows IDE windows that display watch expressions and local variables to work as expected. Current implementation restrictions in the Inspector Agent prevent evaluation of method calls in watch expressions, but all other kinds of watch expressions work as expected.

The Maxine JDWP Server creates *artificial fields* for the transmission of additional, implementation-related information about Java objects. It can do this because the server controls class layout information transmitted to the client. Requests for read access to artificial fields are handled directly by the server, whereas access to other fields requires reading from VM memory via the Inspector Agent. Figure 3 shows an example Java object and how it would appear in the debugger client. At this time the server simulates fields for the address and the header word of an object, both of which are machine-level VM implementation artifacts. The server also simulates a field that points to an object's *hub*: a Maxine VM implementation object describing the type and other meta-data related to the object. The client debugger requires no modification to display artificial fields, since they appear as ordinary Java fields.



Name	Type	Value
p	Point	#70
X_hub	Hub	#71
x	int	2
y	int	8
X_misc	int	0xfff50003
X_address	int	0xffef35b0

Fig. 3. Artificial object fields help transmitting additional information about objects to the debugger.

Shared Interface

```
interface A {
  String foo();
}
```

Implementation on Server Side

```
class AImpl implements A {
  String foo() {
    return “Hello world!”;
  }
}
```

Access on Client

```
// Obtain JDI object reference
ObjectReference reference = getReference();

A a = createProxy(reference, A.class);
String s = a.foo();
```

```
class DynamicProxy implements InvocationHandler {

  // Encapsulated JDI reference
  ObjectReference reference;

  public Object invoke(Object proxy,
                       Method method,
                       Object[] args) {
    ...
    // Send JDWP Invoke command
    ...
  }
}
```

Fig. 4. Code example for the Java dynamic proxy mechanism combined with JDWP.

4 A Protocol within the JDWP Protocol

Although artificial fields permit the display of additional information about objects without modification to the JDWP debugger client, a more general mechanism is also needed. This is done without change to the protocol by leveraging the JDWP `invoke` command, which was originally intended to support method calls in watch expressions. Java's *dynamic proxy* mechanism [10] makes it possible to create proxy objects behind an interface, objects that delegate method calls to JDWP `invoke` commands. The Maxine JDWP Server creates *artificial methods* that provide access to machine-level information via reflective delegation to appropriate methods in the Inspector Agent. The net effect is a kind of specialized "remote method invocation" available to the client through an interface.

Code samples in Figure 4 show how the dynamic proxy mechanism is implemented. Interface `A` is defined on both server and client. On the server it is implemented by class `AImpl`; on the client a dynamic proxy object is created. The client-side proxy implements the interface `InvocationHandler`, which allows the delegation of Java calls; it delegates method calls to the `invoke` method.

Figure 5 diagrams the interaction among objects in this architecture. The client implements the interface with a Java dynamic proxy object; it is based on a JDI Object Reference to which it delegates method calls. This JDI Object Reference is also known to the NetBeans Debugger and can be referenced in JDWP commands. The Maxine JDWP Server delegates `invoke` commands on a specific JDWP ID to the corresponding Java object. In conventional usage, server-side `invoke` commands are delegated to the VM via the Inspector Agent, but in this case they are redirected via reflective call to the implementer of the interface. Glue code for these interactions is automated, so neither the user of the interface nor the implementer need any special handling.

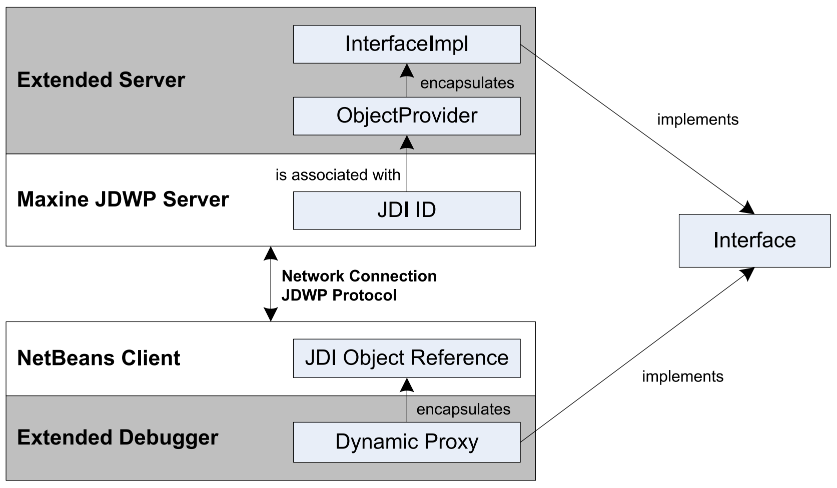


Fig. 5. Dynamic proxy objects for implementing a protocol within the protocol.

Three important optimizations address performance concerns:

- *State Change:* The client normally presumes that a conventional JDWP `invoke` invalidates previously retrieved VM state. We can guarantee, however, that an `invoke` on an artificial method in the Maxine JDWP Server executes no VM code. In these cases Java reflection is used to bypass the built-in refresh mechanisms of JDI, thus avoiding unnecessary overhead on the client.

– *Multiple Objects:* Although transmitting data is typically fast, the round trip needed to perform a JDWP `invoke` is not. Some return values (e.g. the information about all registers of a thread) are represented by multiple objects, and transmitting them individually could introduce undesirable latency. Such values are instead returned from server to client as a unit: a byte array containing the serialized data needed to reconstruct the object graph.
– *Cache:* Many of the interface methods used in this architecture are guaranteed to return the same result when called with the same parameters. The client further optimizes network traffic by caching these results, based on annotations applied to such methods. This grants client side implementations the freedom to invoke remote commands without undue performance concerns.

This approach, by virtue of specialization, has advantages over Java RMI or other Java remote technologies in this context:

– It shares an existing JDWP network connection.
– Interfaces can be used to extend standard JDWP objects. For example, a special interface adds methods to a thread object that provide access to register state.
– JDWP objects can appear as method parameters and return types.

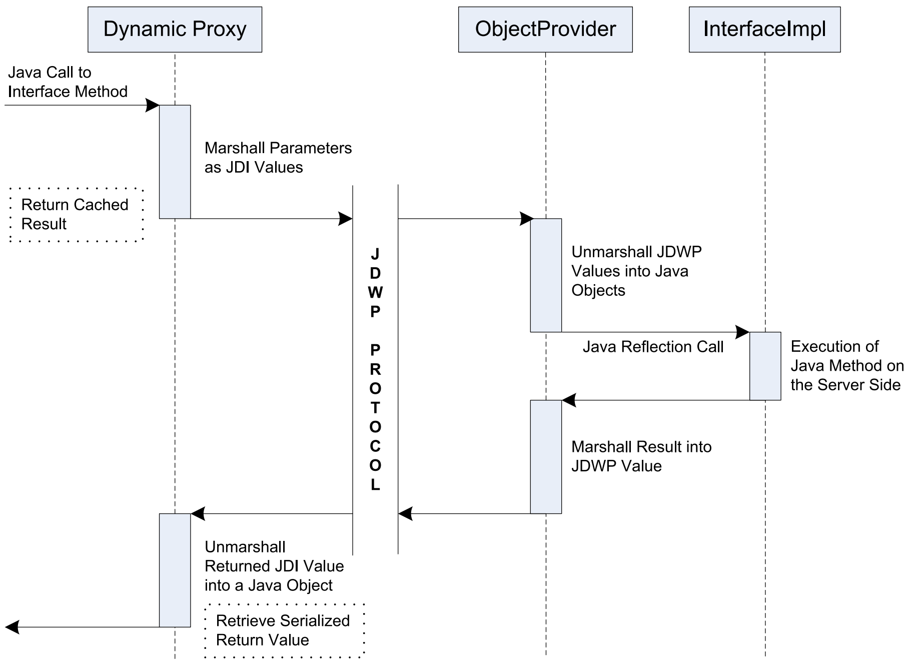
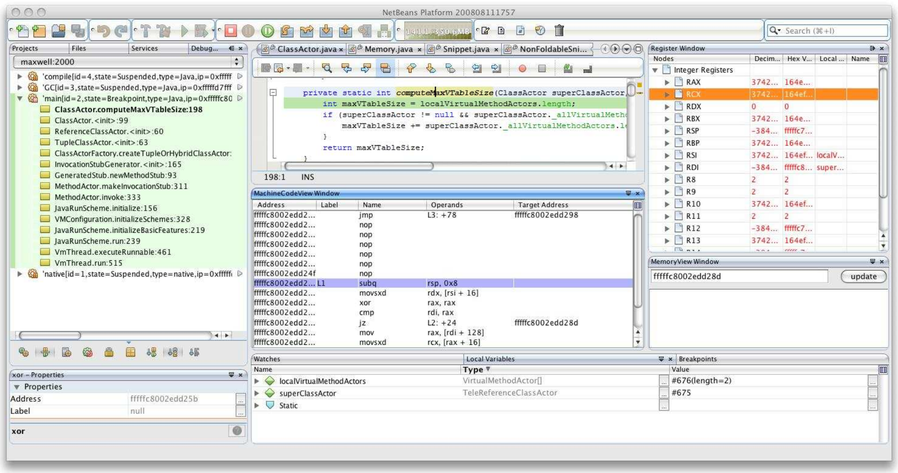

Dynamic Proxy
ObjectProvider
InterfaceImpl
Java Call to Interface Method
Marshall Parameters as JDI Values
Return Cached Result
JDWP PROTOCOL
Unmarshall JDWP Values into Java Objects
Java Reflection Call
Execution of Java Method on the Server Side
Marshall Result into JDWP Value
Unmarshall Returned JDI Value into a Java Object
Retrieve Serialized Return Value

Fig. 6. Example sequence of a call to an interface.

Figure 6 diagrams the interactions that follow when the client invokes an artificial method on the Maxine JDWP Server. The client's dynamic proxy implements the call, checking first whether the result is cached and can be returned immediately. If not cached, the client marshalls parameters as JDI values for transmission via JDWP `invoke`. Primitive types are marshalled by encapsulation in a JDI data object, reference types by JDWP identifiers managed on the server. Array types are simply copied to the server and filled with marshalled primitive or reference values.

The Maxine JDWP Server first unmarshalls the `invoke` parameters, using a map to convert JDWP identifiers to object references. The call is then delegated via Java reflection to the implementer of the interface. Finally the server marshalls the return value into a JDWP value for return to the client.

The client's dynamic proxy receives the return value as a JDI object and converts it to a Java object. In case of an array this can again require additional JDWP commands to retrieve array contents. In case of a byte array containing the serialized form of a Java object, the bytes are retrieved and deserialized. The original caller receives a normal Java object in return without special treatment.



```
'compile[id=4,state=Suspended,type=Java,ip=0xfffff
'GC[id=3,state=Suspended,type=Java,ip=0xfffffd7fff
'main[id=2,state=Breakpoint,type=Java,ip=0xfffffc80
    ClassActor.computeMaxVTableSize:198
    ClassActor.<init>:99
    ReferenceClassActor.<init>:60
    TupleClassActor.<init>:63
    ClassActorFactory.createTupleOrHybridClassActor:
    InvocationStubGenerator.<init>:165
    GeneratedStub.newMethodStub:93
    MethodActor.makeInvocationStub:311
    MethodActor.invoke:333
    JavaRunScheme.initialize:156
    VMConfiguration.initializeSchemes:328
    JavaRunScheme.initializeBasicFeatures:219
    JavaRunScheme.run:239
    VmThread.executeRunnable:461
    VmThread.run:515
'native[id=1,state=Suspended,type=native,ip=0xfffff
```

```
private static int computeMaxVTableSize(ClassActor superClassActor,
    int maxVTableSize = localVirtualMethodActors.length;
    if (superClassActor != null && superClassActor._allVirtualMeth
        maxVTableSize += superClassActor._allVirtualMethodActors.le
    }
    return maxVTableSize;
}
```

198:1 INS

xor – Properties

Properties	
Address	fffffc8002edd25b
Label	null

xor

MachineCodeView Window

```
Address            Label  Name     Operands        Target Address
fffffc8002edd2...         jmp      L3: +78         fffffc8002edd298
fffffc8002edd2...         nop
fffffc8002edd2...         nop
fffffc8002edd2...         nop
fffffc8002edd2...         nop
fffffc8002edd2...         nop
fffffc8002edd24f          nop
fffffc8002edd2... L1      subq     rsp, 0x8
fffffc8002edd2...         movsxd   rdx, [rsi + 16]
fffffc8002edd2...         xor      rax, rax
fffffc8002edd2...         cmp      rdi, rax
fffffc8002edd2...         jz       L2: +24         fffffc8002edd28d
fffffc8002edd2...         mov      rax, [rdi + 128]
fffffc8002edd2...         movsxd   rcx, [rax + 16]
```

Register Window

```
Nodes                 Decim...  Hex V...   Local ...  Name
Integer Registers
  RAX                 3742...   164e...
  RCX                 3742...   164ef...
  RDX                 0         0
  RBX                 3742...   164e...
  RSP                 -384...   fffffc7...
  RBP                 3742...   164e...
  RSI                 3742...   164ef...   localV...
  RDI                 -384...   fffffc8...  super...
  R8                  2         2
  R9                  2         2
  R10                 3742...   164ef...
  R11                 2         2
  R12                 -384...   fffffc7...
  R13                 3742...   164ef...
```

MemoryView Window

fffffc8002edd28d update

Watches | Local Variables | Breakpoints

Name	Type	Value
localVirtualMethodActors	VirtualMethodActor[]	#676(length=2)
superClassActor	TeleReferenceClassActor	#675
Static		

Fig. 7. Screenshot of the enhanced NetBeans debugger showing both machine code and Java source code position.

5 Status and Results

The architecture of our approach makes it possible for any client that implements JDWP to debug the Maxine VM at the Java-level. Multi-level debugging, however, requires additional functionality, for which we chose to extend NetBeans. NetBeans already implements JDWP, has a flexible plugin architecture, and provides standard techniques for extending debugger functionality without modification.

Our approach succeeds in making it simple to transport additional information between the Maxine Inspector Agent and the NetBeans IDE. Client-side code for displaying data can use simple interfaces that are implemented on the server. The Java dynamic proxy mechanism hides the complications implied by data marshalling and unmarshalling to transmit the data over the network.

The Maxine JDWP server implements both a useful subset of the standard source-level JDWP protocol and access to the VM's machine-level state. We have prototyped additional machine-level views; Figure 7 shows NetBeans debugging the Maxine VM with these extensions enabled. The Java-level call stack, local variables, and current position in Java code appear in NetBeans components. The Maxine Code View highlights the current machine code location based on the current position in Java code. Also shown are register contents for the currently selected thread. The address of each Java object appears in an artificial field, and this address can be used to retrieve the raw memory contents at that location.

The full reuse of the NetBeans Java debugger frees us from implementing many concepts that are part of a modern Java debugger, and ensures ongoing benefit as NetBeans evolves. On the other hand, it remains to be seen whether the tight integration among views at different levels of abstraction will be as easy as it has been in the specialized Maxine Inspector. Implementing the prototype described here has already required the use of Java reflection in order to access NetBeans functionality for which no public API is provided. This difficulty is not a limitation of the transport architecture described in this paper, but rather the nature of an IDE for which this type of extension was perhaps not anticipated.

The prototype described here is integrated into the Maxine open source project. We plan to continue exploring this approach to multi-level debugging for the Maxine VM with work on additional components and mutli-level views.

6 Related Work

Ungar et al. developed a meta-circular VM for the Self programming language [15], which they debugged with a debug server written in C++. Remote debugging in Self relies upon a custom network protocol as well as a custom debugger.

The meta-circular Jikes Research Virtual Machine is also written in Java [1]; it uses a remote reflection technique for accessing VM data that is similar to ours [7]. The Jikes debugger itself however, is not based on an existing Java debugger.

Simon et al. [9] developed the Squawk VM, a small Java VM written mostly in Java that runs on a wireless sensor platform. Squawk includes a debug agent that implements a subset of JDWP, but there are no extensions of the sort that permit multi-level debugging.

Printezis and Jones created GCspy framework for the analysis of VM memory management behavior [8]. They considered and rejected using an enhanced version of JDWP for transmitting their data for two reasons: GCspy data has nothing to do with debugging, the focus of the JDWP protocol, and its use would needlessly confine GCspy technology to Java applications. These, however, are actually advantages in the context of Maxine debugging.

Dimitriev extended the JDWP protocol with additional commands for redefining classes in the VM [2][3]. Although this approach is reasonable for functionalities that are of general interest for virtual machines, it would complicate the protocol significantly in our case. By defining a protocol within the JDWP protocol, we retained flexibility and avoided the need for further standardization.

7 Conclusions

Multi-level debugging for a meta-circular VM presents technical challenges that are not easily met with conventional development tools, even mixed-mode tools designed to support multiple languages. Faced with the choice between the development cost of a specialized, isolated tool (our original approach), and the challenges of directly extending an existing IDE, we have explored an alternate approach. Leveraging a single-level standard architecture (JPDA) for remote debugging, we were able to transport information at additional levels of abstraction by implementing a remote invocation protocol within the JPDA's Java Debugging Wire Protocol (JDWP). The advantages of this approach, prototyped using the Maxine Inspector Agent on the server side and NetBeans as the client, are that a wide array of rich functionality becomes available for debugging the Maxine VM at the Java-level. This dramatically reduces Maxine development cost for this level of functionality, both present and future as the NetBeans platform evolves.

Solving the transport problem, however, is only part of the solution. The next challenge is to provide the advantages of tightly integrated multi-level views, easily developed in the stand-alone Maxine Inspector, in a rich platform that was originally designed for single-level debugging.

8 Acknowledgements

The authors would like to thank Maxine team members Ben Titzer and Bernd Mathiske for their support and many helpful suggestions.

References

1. Alpern, B., Attanasio, C.R., Cocchi, A., Hummel, S.F., Lieber, D., Mergen, M., Shepherd, J.C., Smith, S.: Implementing jalapeño in Java. In: OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press (1999) 314–324
2. Dmitriev, M.: Safe class and data evolution in large and long-lived Java^{TM} applications. Technical report, Sun Microsystems Laboratories (2001)
3. Dmitriev, M.: Towards flexible and safe technology for runtime evolution of Java language applications. In: Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference. (2001)

4. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java^{TM} Language Specification, The (3rd Edition) (Java Series). Addison-Wesley Professional (2005)
5. Liang, S.: The Java Native Interface: Programmers Guide and Specification. Addison-Wesley Publishing Co., Inc. (1999)
6. Mathiske, B.: The maxine virtual machine and inspector. In: OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications, ACM (2008) 739–740
7. Ngo, T., Barton, J.: Debugging by remote reflection. In: Proceedings of Euro-Par 2000 - Parallel Processing, Springer LNCS (2000) 1031–1038
8. Printezis, T., Jones, R.: Gcspy: an adaptable heap visualisation framework. In: OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM (2002) 343–358
9. Simon, D., Cifuentes, C., Cleal, D., Daniels, J., White, D.: Java^{TM} on the bare metal of wireless sensor devices: the squawk Java virtual machine. In: VEE '06: Proceedings of the 2nd international conference on Virtual execution environments, ACM (2006) 78–88
10. Sun Microsystems, Inc.: Java Dynamic Proxy Classes. (1999) http://java.sun.com/j2se/1.5.0/docs/guide/reflection/proxy.html.
11. Sun Microsystems, Inc.: Java Debug Interface. (2004) http://java.sun.com/javase/6/docs/jdk/api/jpda/jdi/index.html.
12. Sun Microsystems, Inc.: Java Debug Wire Protocol. (2004) http://java.sun.com/javase/6/docs/technotes/guides/jpda/jdwp-spec.html.
13. Sun Microsystems, Inc.: Java Platform Debugger Architecture. (2004) http://java.sun.com/javase/6/docs/technotes/guides/jpda/.
14. Sun Microsystems, Inc.: NetBeans. (2009) http://www.netbeans.org.
15. Ungar, D., Spitz, A., Ausch, A.: Constructing a metacircular virtual machine in an exploratory programming environment. In: OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM (2005) 11–20