Multilanguage Debugger Architecture

Jan Vraný and Michal Píše

Department of Computer Science and Engineering
Czech Technical University in Prague
Faculty of Electrical Engineering

Abstract. When debugging applications written in several programming languages, debuggers fail to provide programmers with the same quality of user experience that is common for single-language applications debugging. Therefore, the development of multiple-language applications tends to be more expensive both in terms of development costs and maintenance necessary in the fu
In this paper, we provide a description of a debugger architecture that is capable of integrating debugging information from multiple language runtime e provide the same quality of user experience as any single-language debugger. The described architecture has been tried out on a proof-of-concept im allows debugging of applications written in Smalltalk, XQuery and JavaScript.

1 Introduction

Nowadays, applications written in several programming languages are not uncommon. For example, typical web application's frontend uses some form of a template engine with limited scripting capabilities, its business logic is implemented using an object-oriented language such as Java or C# and its persistence layer composes of a set of triggers and stored procedures written in a SQL dialect.

The reasons for utilizing multiple languages within a single application vary: first, some parts of the application's logic are sometimes much easily and accessibly expressed in a different language then the rest of it. Such a situation leads either to creation of a relatively small scripted language that falls within the category of domain specific languages [Mernik et al., 2005] or to usage of a general purpose scripting language such as Python or JavaScript. Second, it might be efficient in terms of time and programmers' effort to develop parts of an application in different languages because of already existing reusable codebase, libraries or suitable frameworks. Third, a low-level language (usually C) routines are often embedded into other languages for performance reasons and finally, utilization of several languages in one application may simply be the only technologically available option (*e.g.,* when implementing triggers in an Oracle database, Java or PL/SQL has to be used even though the rest of the application is written in PHP).

Debugging such heterogeneous applications brings difficulties for developers, especially when debugging code composed of chunks written in different languages, as debuggers usually support only one language. For instance, consider a Java application that uses a native class (*i.e.,* invokes embedded C functions). When debugging

this application, developers have three basic options. First, they may use a Java debugger, which means they will be able to inspect input parameters and return value of any embedded C function but they will not be able to debug bodies of the C functions themselves. Second, they may use a C/C++ debugger—that way they will be able to debug bodies of C functions as well but they will also have to step through layers of virtual machine implementation code when debugging Java code. Finally, they may decide to use Java debugger to debug Java code and C/C++ debugger to debug embedded C code but they will have to manually start/suspend each debugger when entering code written in the respective language and they will have to merge information provided by the two user interfaces themselves which is hardly convenient or effective.

To sum it up, when debugging multiple-language applications, debuggers fail to provide programmers with the same quality of user experience that is common for single-language applications debugging. Consequently, debugging of multiple-language applications is more resource-costly and error-prone[1]. Naturally, a debugger that provides the same quality of user experience regardless of whether the application is single- or multiple-language is highly desirable.

The aim of this paper is to propose architecture of such debugger. Namely, its contributions are as follows: (i) description of a flexible source-level debugger architecture that facilitates integration of multiple single-language debuggers into a single unified debugging environment that allows debugging of multi-language applications in their entirety and (ii) proof-of-concept implementation that allows debugging of applications written in Smalltalk, XQuery and JavaScript.

The structure of this paper is the following: section 2 provides the necessary background, section 3 described the proposed solution, section 4 compares our solution with the related work and section 5 concludes the paper.

2 Background

This section contains a brief description of how a new language may be implemented, what are the possible approaches to utilization of multiple languages within one application and what are the possible ways of implementing a source-level debugger.

2.1 Language Implementation

An implementation of a language usually consists of two parts: a compiler and an execution runtime. The compiler's role is to transform the source code of an application into a representation which is understood by its respective execution runtime: machine code, byte code or abstract syntax tree (although in some cases, the runtime executes the source code directly, which eliminates the need for compiler).

The role of the runtime is then to perform the computation by interpreting the intermediate program representation created by the compiler. In certain cases, there is no need for execution runtime since the compiler generates machine code that runs on

[1] Which is especially piquant since debugger is supposed to ease and speed up removal of bugs – not to complicate it and slow it down.

bare hardware. However, with the progression of virtual machines technology and, at the same time, increasing complexity of standard libraries used in languages compiled into native code, the distinction between pure interpretation and direct machine code execution is getting more and more blurred.

2.2 Multiple-language Applications

There exist three basic ways of mixing two or more languages in a single application. The first is an interpreter written in one of the languages interpreting the other language. In such case, invocation of a method written in the second language from a code written in the first language may be achieved by calling the interpreter with the method identification supplied as its argument. In other words, developer invokes the interpreter which in turn invokes the method written in the other language.

This approach is usually used for execution of small pieces of code written in some specialized DSL and is often based on the interpreter design pattern [Gamma et al., 1993]. Its advantage is that it is easy to implement, modify and extend. However, its performance tends to be poor.

The second way is to reuse an existing runtime and let the compiler produce intermediate code representation (usually bytecode) understood by that runtime while emulating semantics that is not available on the target platform. For example, the JRuby[2] compiler compiles Ruby code to the Java Virtual Machine (JVM) bytecode, which is then interpreted by the JVM.

Such approach is very effective in terms of language implementor's effort, usually provides fair performance and the integration of the two languages is almost seamless in both ways. However, it is suitable only for languages with fairly similar basic notions—for example it would probably be very difficult to use this approach to integrate a prototype-based language with continuations and no notion of call stack with a runtime designed for a language based on logic programming paradigm.

The third way is to pass the execution to a native code. This approach is both efficient and flexible, however, it requires for the calling language to have some kind of support for invocation of methods written in another languages. In other words, the language must provide some mechanism of stepping out of its runtime, converting all necessary data (parameters and return value) and invoking the runtime of the other language. Obviously, it is not an option that is always viable.

2.3 Debuggers and Their Implementations

Debuggers are developers' tools which ease finding and removal of bugs in the code. Typically, debuggers allow the programmer to stop the debugged program's execution when it reaches one of the previously designated statements in the code (breakpoints), execute the program's expressions and statements one by one in their respective order (we call this a *debugging operation*) and inspect the program's state, i.e. variables on the stack and on the heap (*execution state*).

[2] http://jruby.codehaus.org

There are two basic means of collecting data about which expression or statement is currently under execution: code instrumentation and events emitted by operating system, virtual machine or interpreter [Lencevicius, 2000].

When using code instrumentation, the application's code is augmented to contain method calls that inform their recipient of the currently executed expression. The code may be instrumented statically—by the compiler or some kind of code instrumentation tool—or dynamically—when it is being loaded into the memory or even before its first execution [Hofer, 2006].

The second approach differs from the first one in that the events are emitted by the interpreter, virtual machine or operating system [Wu et al., 2005]. In other words, debugging-related instructions are part of the interpreter, virtual machine or operating system, not the application itself.

To map program's execution state to its source code, debuggers make use of debug symbols. Debug symbols are metadata enabling the debugger to gain additional information about the code—the names of the methods and variables, mapping from the instruction to the lines of code etc. They are generated by the compiler and can be distributed either with the compiled code or in a separate file.

3 Solution

This section describes architecture of unified debugger – a new debugger implementation capable to debug multi-language application.

3.1 Overview

The architecture of the unified debugger consists of several components that communicates through event mechanism. Figure 1 shows core components of the debugger.

DebuggerAdapter. An debugger adapter is a core class that acts as a facade for underlying execution engine. It is used for both accessing the control flow structures such as contexts and variables and for controlling the execution. Basically there is one debugger adapter for each language.

ContextAdapter, VariableAdapter and InstructionAdapter. These are helper classes, that provide uniform access to language's execution engine internals.

DebuggerService. A debugging service is responsible for performing debugging operations such as *step-into* or *step-over*. It's a mediator between the debugger adapter and a debugger user interface.

Mode and subclasses. Debugger service modes represent a debugging action to be taken next time the interpreter executes a piece of code.

DebuggerUI. Finally, the debugger is the user interface. It presents the source code and the execution state to the programmer. It also enables the programmer to perform debugging operations from the UI.

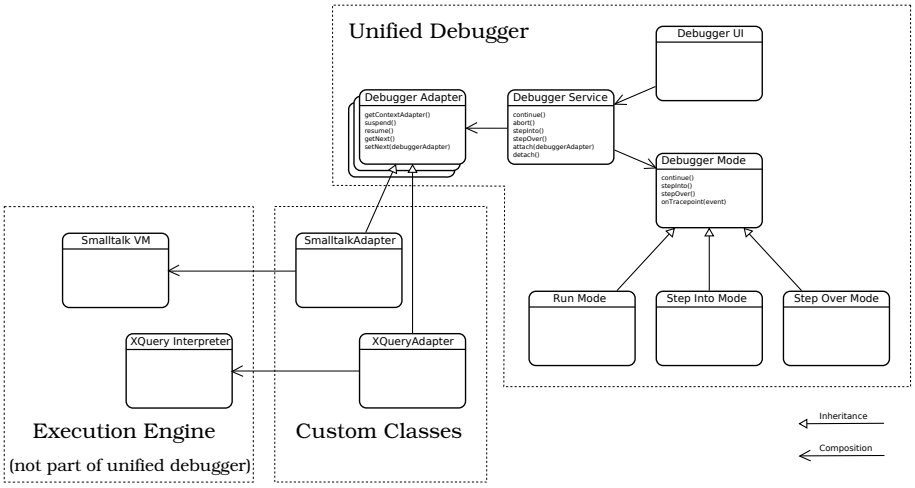

Fig. 1: Overall architecture of unified debugger

3.2 Debugger Adapter

As we said before, the debugger adapter is a core class of whole system. It interfaces underlying execution engine in general way, no matter how it is implemented. More precisely, the debugger adapter (i) provides an uniform access to the current execution state, (ii) emits events whenever the execution state of the program changes and (iii) provides facilities for suspending/resuming the execution. Generally, there is one debugger adapter for every interpreter. In practice, debugger adapter implementations can be shared between several languages that are implemented in same way.

For example, the PERSEUS framework[3]. provides a rich set of reusable classes for building language interpreters with integrated debugging facilities based on debuggable interpreter design pattern [Vraný and Bergel, 2007].This framework also provide a debugger adapter implementation that can be used for all languages based on the framework.

Execution state model

The execution state modeled through set of adapters: a context adapter, a variable adapter and an instruction adapter. The adapter objects are used by the user interface to present the state to the programmer.

Context adapter. Context stack is model-led by context adapters. Each context adapter belongs to one activation record on interpreter's execution stack. The context adapter provides an access to:

– name of the function or method that belongs to the context,

[3] http://www.squeaksource.com/Perseus.html,
http://smalltalk.felk.cvut.cz/projects/Perseus

– source code of that function or method,
– context adapter of the sender (caller) context (as another instruction adapter),
– instruction being interpreted (as instruction adapter),
– set of variables that belongs to the context (as variable adapters).

The adapter also contains a reference to the debugger adapter it belongs to.
Instruction adapter. The instruction adapter represents an instruction being interpreted. It contains a line reference to the source code, which is used by the debugger to visually emphasize current position in the code. Although this object is called instruction adapter, it is generally not related to the interpreter's (or hardware processor's) program counter register. Here the instruction is just an abstraction of the smallest piece of code that is executed atomically by an interpreter. An instruction might be a single bytecode or an AST node, depending on interpreter's internal architecture.
Variable adapter. Variable adapters abstracts function arguments and local variables. It also enables the debugger read and modify variable's value.

Although presented set of adapters covers wide range of programming languages, it does not completely cover all possible languages. A new kinds of adapters and properties can be easily added using customized adapters.

Events Emitted by the Debugger Adapter

During a program execution the debugger adapter emits number of events. These events, called *announcements*, reflect changes in program's execution state such as entering or leaving a function, modifying a variable or the reach of a breakpoint. Announcements allows other objects such as debugger service to analyze interpreter's control flow an react whenever certain situation occurs. Figure 2 shows hierarchy of announcements.

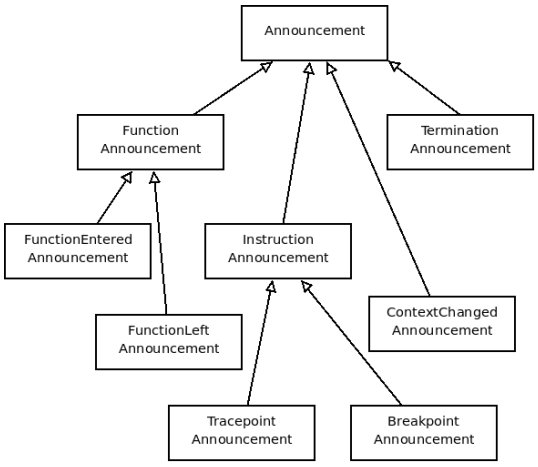

Fig. 2: Announcement class hierarchy

Execution Control Facilities

During an interactive debugging session, the program execution is interlaced with debugging phase. In debugging phase, the program execution is temporarily suspended and programmers are given a chance to interactively explore and modify program state such as variable values. At the end of debugging phase, programmers might resume program execution by means of debugging operation or abort the execution at all.

To enable interactive debugging, the debugger adapter expose three methods with obvious meaning: *suspend*, *resume* and *abort*. Those method are used by the debugger service to drive the execution during an interactive debugging session.

3.3 Debugger Service

The debugger service implements the debugging operations such as step-into, step-over and continue. It acts as a model for debugger user interface. During a debugging session the debugger service is *attached* to the debugger adapter. That means that the debugger service is registered to the adapter and receives emitted announcements:

```
DebuggerService>>attach: anDebuggerAdapter {
  adapter ← anDebuggerAdapter.
  adapter
    subscribe: TracepointAnnouncement
      send: #onTracepoint: to: self;
    subscribe: ContextAnnouncement
      send: #onContextChange: to: self;
}
```

At the end of debugging session the debugger service *detaches* from the debugger adapter:

```
DebuggerService>>detach {
  adapter unsubscribe: self.
}
```

For the more detailed description of the debugging service and debugging operation implementations please refer to [Vraný and Bergel, 2007].

3.4 Stacking Debugger Adapters

The basic idea of our unified debugger is following: in addition to to the execution stack, a stack of debugger adapters is maintained. Each debugger adapter corresponds to a bunch of activation records on the execution stack.

Debugger adapter stack must be maintained manually (*i.e.,* programmer should include stack modification code into the code) whenever a program's control flow enters or leave a chunk of code in different language than the one currently being executed. Two functions are provided for managing interpreter adapter stack: *pushDebugger-Adapter:* and *popDebuggerAdapter*. When a new debugger adapter is pushed onto a stack, all debugger services that are attached to a current debugger adapter must attach a new debugger adapter:

```
DebugggerAdapter class>>
 pushDebuggerAdapter: newDebuggerAdapter {
  activeAdapter subscribers do:
    [:subscriber|
    subscriber detach.
    subscriber attach: newDebuggerAdapter].
  newInterpreter next: activeAdapter.
  activeAdapter ← newDebuggerAdapter
}
```

Similarly when the topmost adapter is to be removed, all attached debugging services must reattach the next one:

```
InterpreterAdapter class>>popDebuggerAdapter {
  activeAdapter subscribers do:
    [:subscriber|
    subscriber detach.
    subscriber attach: activeAdapter next].
  activeAdapter ← activeAdapter next
}
```

Manual management of debugger adapter stack is usually not big deal since calling foreign functions (that is routines implemented in another language) often requires some glue code.

Consider a following example of multi-language applications written in Smalltalk, XQuery and JavaScript. Smalltalk part of the application instantiates an XQuery interpreter and evaluates XQuery code:

```
|xqInterpreter |
xqInterpreter ← XQueryIntepreter new.
xqInterpreter evaluate: query
```

The `query` variable holds an XQuery code, that defines new function for computing combinatorial numbers:

```
import module namespace js = "http://sma...";
declare function combinatorial-number ( $n , $k ) {
  js:factorial( $n ) /
    ( js:factorial ( $k )
      * js:factorial ( $n - $k ) )
};

combinatorial-number ( 5 , 3 )
```

The XQuery code calls a function `js:factorial`, that is implemented in JavaScript:

```
function factorial ( a ) {
  if ( a == 0 ) {
    return 1;
  } else {
```

```
    return a * factorial ( a );
  }
}
```

Figure 3 shows execution and debugger adapter stack for the example above.

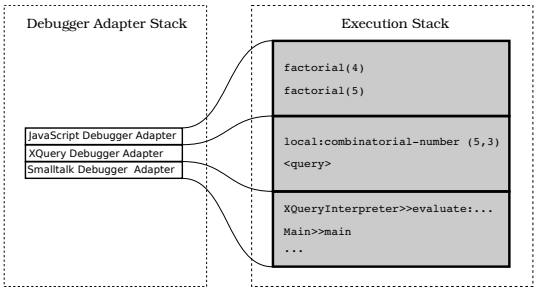

Fig. 3: Debugger adapter stack

The method *evaluate:* in unified debugger-enabled XQueryInterpreter class is – on principle – implemented as follows:

```
XQueryInterpreter>>evaluate: query {
  | queryTree result |
  queryTree ← self parse: query.
  DebuggerAdapter pushDebuggerAdapter:
    (XQueryDebuggerAdapter on: self).
  result ← self visit: queryTree.
  InterpreterAdapter popDebuggerAdapter.
  ↑ result.
}
```

The XQuery interpreter supports number of primitives. Primitives are functions that are directly callable from within an XQuery code, but whose implementation is done in different language than XQuery. The XQueryInterpreter calls method *performJsPrimitive:withArguments:* to call primitive implemented in JavaScript:

```
XQueryInterpreter>>evaluateJsPrimitive: primName
 withArguments: args {
  | result |
  DebuggerAdapter pushDebuggerAdapter:
    (ByteCodeDebuggerAdapter on: self).
  result ← jsPrimitiveLibrary
    perform: primName withArguments: args.[4]
  DebuggerAdapter popDebuggerAdapter.
  ↑ result.
}
```

The figure 4 shows a user interface for our unified debugger implementation. A user can explore the stack and resume the evaluation in both step-into and step-over manner.

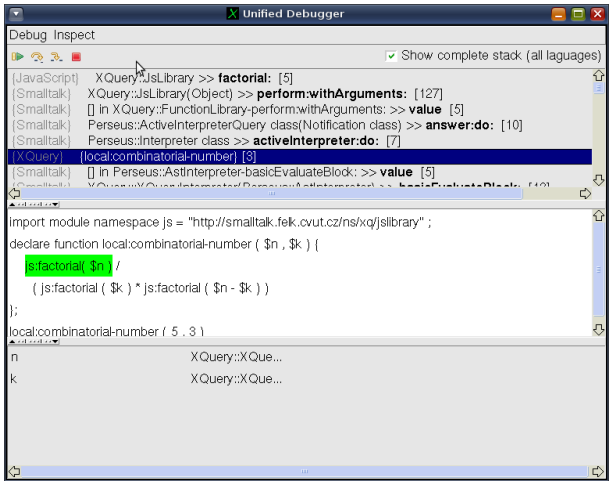

Fig. 4: Unified Debugger User Interface

4 Related Work

NetBeans and Eclipse are integrated development environments (IDEs) used mostly for the development of Java applications. Since they have been designed with generality in mind, plugins for other languages (e.g. Groovy or Ruby) are readily available.

From the outsider's point of view it seems that the architecture of debugger subsystems in both these IDEs has had two main design goals: to enable smooth pluggability of new debuggers and to facilitate code reuse. Using our terminology, their architecture is composed of debugger adapter sending events directly to user interface. User interface code is the same for all debugger adapters, which means once a debugger for a given language has been developed it is relatively easy to write bindings that enable its usage in NetBeans or Eclipse.

NetBeans IDE is capable of debugging multiple-language applications as long as all of the debugged languages are compiled into the Java bytecode. It uses class file debug symbols that map individual bytecode instructions to tuple (file name, line number). In other words, the NetBeans debugger makes use of standard JPDA[5] interface not knowing whether the (byte)code currently under execution has been compiled from Java, Grovy, Ruby or Python.

This approach has one advantage: a new language running on top of the JVM can be debugged immediately (assuming its compiler generated correct debug symbols). On

[5] http://java.sun.com/javase/technologies/core/toolsapis/jpda/

the other hand, this approach implies the debugger is aware only of the JVM metamodel and can not accurately display information that goes beyond this metamodel—for example, it is completely oblivious to Python's dynamically added instance variables. Because of this reason, we consider such approach not scalable enough in the long term.

5 Conclusion and Future Work

Debugging of multiple-language applications is hard and current debuggers offer little help to alleviate the problem. The only debuggable multiple-language applications are those composed of languages compiled to the same runtime, *e.g.,* applications composed of C and C++ code, applications composed of Java, JRuby and Jython code etc.

As a remedy, this paper proposed an architecture that is able to merge multiple single-language debuggers into a unified one capable of debugging multiple-language applications. The architecture introduces a new abstraction layer—the debugger adapter—which interfaces with underlying execution runtimes. Stack of debugger adapters then manages transition of control from one debugger adapter to another.

Validity of this architecture is based on two assumptions: (i) it is always possible to intercept a message invocation and (ii) it is always possible to tell the language of the method that is about to be invoked. The first assumption is always fulfilled as all real-world lanugages do have debuggers capable of emitting event on method dispatch. The second assumption is fulfillable easily: compilers of each language used in the application only need to generate a debug symbol (or an annotation) denoting the language of the currently compiled code.

We believe that the architecture is easily extensible to merge debug information from debuggers running in different processes. That makes it a good candidate for debugging of RPC-based applications as well as systems in which method invocation results in new process creation (such as shell scripts). In future, we also plan to integrate the unified debugger into an industry strength IDE such as NetBeans or Eclipse.

Bibliography

[Gamma et al., 1993]Gamma, E., Helm, R., Vlissides, J., and Johnson, R. E. (1993). Design patterns: Abstraction and reuse of object-oriented design. In Nierstrasz, O., editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 406–431, Kaiserslautern, Germany. Springer-Verlag.

[Hofer, 2006]Hofer, C. (2006). Implementing a backward-in-time debugger. Master's thesis, University of Bern.

[Lencevicius, 2000]Lencevicius, R. (2000). *Advanced Debugging Methods*. Kluwer Academic Publishers.

[Mernik et al., 2005]Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344.

[Vraný and Bergel, 2007]Vraný, J. and Bergel, A. (2007). The debuggable interpreter design pattern. In *In Proceedings of the International Conference on Software and Data Technologies (ICSOFT 2007)*.

[Wu et al., 2005]Wu, H., Gray, J., Roychoudhury, S., and Mernik, M. (2005). Weaving a debugging aspect into domain-specific language grammars. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1370–1374, New York, NY, USA. ACM Press.