Scripting Runtime Dynamic Analyses

Jonathan E. Cook Abdulmalik Al-Gahmi Shalini Devi Navin Vedagiri
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003 USA
jcook@cs.nmsu.edu

Abstract

Large scale system development and maintenance projects often need to build scaffolding—tools that help build the target system—that is customized to the project. For some classes of tools, including dynamic analysis, the cost barrier is too high to consider implementing customized support that might be beneficial to the project, and thus the project makes do with whatever off-the-shelf support is available.

This paper presents ideas and prototypes in offering generic support for high-level, flexible, and programmable introspection of software systems. Our hypothesis is that "quick-and-dirty" scripting languages such as Tcl/Tk and Python can be effectively used to create ad-hoc dynamic analyses that help system engineers better understand, develop, and maintain their system.

1. Introduction

Many system development and maintenance activities need or can benefit from introspective and possibly even manipulative capabilities in a running system. By this we mean the ability to peer into a running system and observe it, and even manipulate it to some extent. But typical mechanisms for introspection are hard to use, involve a great deal of low-level programming, and require expert programming to be used correctly. Because of this, the effort in building introspection tools is very high, and projects are often prevented from building application-specific tools or rapidly prototyping new general-purpose tools.

It would seem natural to provide some generic and easy-to-use mechanism to support these needs, and that is precisely the point of the ideas described here.

Our vision is to provide a flexible, easy-to-use mechanism for introspection that allows not just complex tools to be developed but allows the application programmers to easily build ad-hoc tools that meet a specific need at a specific time. Rather than try to predefine the capabilities *we* think might be needed, a better approach to achieve this end is to re-use one of the many scripting languages that are available.

Scripting languages allow extremely rapid development of functionality, at the the cost of speed since they are interpreted languages. But since they are full programming languages, there is no limit to the type of tools that might be built using them. While they do have some downsides, they seem ideal for building the scaffolding-type of software tools that must be built to help manage, test, observe, and maintain a large production software system.

In our initial prototypes we chose the Tcl/Tk scripting language because of its clean design, ease of integration with traditional programming languages (C/C++), and GUI capabilities. However, the principles underlying our approach can be applied using other languages.

2. Framework

Runtime issues in dynamic analysis have always had to balance the low-level issues of how to instrument the system under observation with the high-level issues of how to make the customization of analysis accessible to the user. A variety of solutions have been proposed and implemented, from special purpose systems that only allow a specific class of analyses to be performed, to special languages (e.g., event processing languages such as [1]) that can be used to specify the desired analysis.

All of these have their place; however, eventually one must consider that the scope of desired dynamic analyses is, in the most inclusive sense, general computation. Thus, why not enable general computational

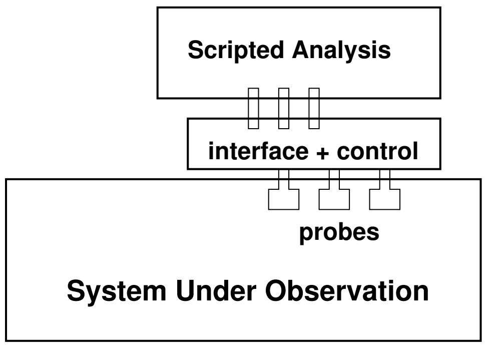


Figure 1. Scripted dynamic analysis architecture.

environments for dynamic analysis? Furthermore, can we make this programming of dynamic analyses more accessible by using high-level programming ideas?

Figure 1 shows how these ideas might fit together. The system under observation should have some mechanism, or at least the potential to insert a mechanism, for observing its behavior. The framework implementor can do the hard task of building the probes on top of this mechanism so that dynamic analysis tools can be built. To hide this complexity, the probe points and information they provide are made available to a scripting language engine, so that a specific analysis can be written in a scripting language, without needing to reach down into the details of the instrumentation.

Our initial prototypes, described in Sections 4-r̃efsec:java, have focused on method/function invocation interceptions, but our idea for the basic architecture is to enable script-level access to more types of instrumentation probes.

3. Tcl/Tk and other scripting languages

Tcl (Tool Command Language [8], pronounced "tickle") is a programming language in the class known as "scripting" languages. Newer scripting languages such as Tcl, Perl, Python, and PHP are much more advanced than the old shell scripting languages, yet they retain the ease of use and the capability for extremely rapid development of advanced functionality. Tcl and most other scripting languages can be both easily executed from C/C++ and extended with custom commands written in C/C++. It is rather misleading to call these languages "scripting" languages, in that they are very powerful interpreted languages, with built-in data structures and functional-style programming language constructs. Modules provide canned support for web services, GUI interfaces, email, ftp, encryption, and many other high-level abstractions.

The upside of scripting languages is that one can create a great deal of functionality with relatively little effort, and they are robust enough to be relied upon. Indeed they can be found running much of the web services we use every day, are used extensively as the foundations of test harnesses, rapid prototyping environments, and many other real world situations.

The downside to most scripting languages is that they do not have a formal semantics but rather an operational one, which can change based on the version of the interpreter one is running! They are targeted towards achieving practical usefulness, not theoretical semantic correctness. However, compiled languages often reveal similar ambiguities [4]. Scripting languages are also quite a bit computationally slower than system programming languages, and their typically weak typing is sometimes detrimental.

4. Realization in CORBA

The CORBA (Common Object Request Broker Architecture) standard has defined cross-platform remote object invocation for ten years [7]. From early on CORBA had a proposed specification for object request interceptors, but it was incomplete and optional. Coincident with version 2.4.2 and later versions, a new interceptor specification was drafted, known as Portable Interceptors [2]. With the Portable Interceptor standard, it is now possible to create debugging, monitoring, and other introspection tools that will interoperate with most vendor ORBs.

In our work, we built an intermediate interceptor layer that took each interception point and invoked a mirror in the scripting language Tcl/Tk. Although not completely invisible to the application developer—some CORBA implementations may require rebuilding the application with different options—there is no low-level programming needed, and CORBA analysis tools that use interception-based data can be written completely in an easy-to-use scripting language.

CORBA Portable Interceptors are ORB-level interceptors that act upon method invocation requests and replies. On the server side, the most basic interception points are "receive_request" and "send_reply". Thus, CORBA interception points naturally give the analysis tool access to the pre- and post-execution points of an invocation. Also note that the interception points are generic for the ORB rather than specific to the object and method being invoked. Our interface makes available to the scripting language an object ID, the method name, and the values and types of the parameters and return values. Thus, at the script level, the analysis can perform computations specific to the object and/or method, by inspecting the meta-data.

5. Realization in Unix shared libraries

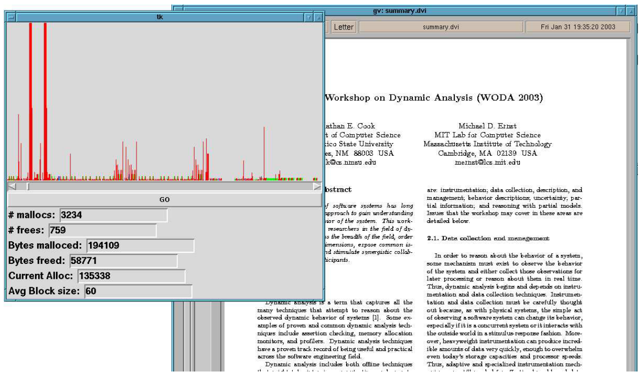
Shared or dynamic link libraries offer an interesting deployment opportunity for our ideas, because they are so widely used and their components are relatively simple (C functions). Nevertheless, the environment is one where very little meta-information is available, and with almost no runtime meta-programming ability. It has potential access, though, because we can modify the dynamic loading process to allow the possibility of binding a function call not to the original target function but to whatever we want, namely a probe point.

Once we have the call intercepted, it is "only" a matter of programming to implement the relay of the function call to the scripted dynamic analysis, and to ensure that the original function is still called, to effect the correct execution of the program. While not trivial, it is possible. Our work is currently in the context of the ELF object and library file formats [6], and in the Gnu shared library loader [5] as used in the Linux operating system.

When creating an object file which has calls to functions located in shared libraries, the compiler produces a call that uses table-based indirection (we will call them "jump tables"). In a somewhat simplified scenario, this table entry initially points not to the actual function (since its location is not known) but to the dynamic loader. Thus the first call invokes the dynamic loader, which will look up the function (by name), load the library if necessary, figure out the actual address of the function being called, overwrite the table entry with the actual function address, and then jump to the function. All subsequent calls from that call site simply pay a tiny (one instruction) penalty of a table lookup.

Although a static name interception capability already existed with the LD_PRELOAD environment variable, we have modified the Gnu dynamic loader to enable dynamic control of the name resolution process, for this and other work we are doing. The dynamic control allows runtime remapping of names to alternative names, on a per-link-object basis rather than at a global level. This functionality gives us the basic interception capability.

To avoid the necessity of low-level programming of the probes, we created a wrapper generator that takes function prototype definitions and generates probe wrappers that the dynamic linker can safely redirect the execution to. These wrappers also instantiate the argument and return data into Tcl-accessible data, and



mallocs: 3234
frees: 759
Bytes malloced: 194109
Bytes freed: 58771
Current Alloc: 135338
Avg Block size: 60

Workshop on Dynamic Analysis (WODA 2003)

athan E. Cook
t of Computer Science
ico State University
es, NM 88003 USA
k@cs.nmsu.edu

Michael D. Ernst
MIT Lab for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139 USA
mernst@lcs.mit.edu

bstract

f software systems has long pproach to gain understanding ior of the system. This work- researchers in the field of dy- s the breadth of the field, order imensions, expose common is- nd stimulate synergistic collab- icipants.

Dynamic analysis is a term that captures all the many techniques that attempt to reason about the observed dynamic behavior of systems [1]. Some examples of proven and common dynamic analysis techniques include assertion checking, memory allocation monitors, and profilers. Dynamic analysis techniques have a proven track record of being useful and practical across the software engineering field.

Dynamic analysis includes both offline techniques

are: instrumentation; data collection, description, and management; behavior descriptions; uncertainty; partial information; and reasoning with partial models. Issues that the workshop may cover in these areas are detailed below.

2.1. Data collection and management

In order to reason about the behavior of a system, some mechanism must exist to observe the behavior of the system and either collect those observations for later processing or reason about them in real time. Thus, dynamic analysis begins and depends on instrumentation and data collection techniques. Instrumentation and data collection must be carefully thought out because, as with physical systems, the simple act of observing a software system can change its behavior, especially if it is a concurrent system or it interacts with the outside world in a stimulus-response fashion. Moreover, heavyweight instrumentation can produce incredible amounts of data very quickly, enough to overwhelm even today's storage capacities and processor speeds. Thus, adaptive and specialized instrumentation mech-

Figure 2. Memory usage analysis in the shared library framework.

invoke Tcl routines before and after the original function is invoked, so that pre- and post-execution analysis can be performed.

Thus, a dynamic analysis of an existing system can be written completely in Tcl, except for the function prototypes needed to generate the wrappers. Figure 2 shows an simple memory allocation analysis of an existing binary executable (Ghostview) that was written purely in Tcl/Tk.

An interesting difference between the interception points in this framework and in the previous one (CORBA) is that the C shared library interception points are specific to each function that is being intercepted. At the scripting level, a procedure must be defined for each pre- and post-interception point, for each function being intercepted. This is quite different than the generic interception point offered by the CORBA. The tradeoff between the two is that specific interception points offer more direct access to perform very specific ad-hoc analyses, while general needs such as event logging are much easier with the generic interception points (and the appropriate meta-data).

We have devised a mechanism for generic interception points in the shared library framework, but are still in the process of implementing it, and need to do more testing and make more meta-data available to define the actual interception.

6. Realization in Java

In both the CORBA and shared library environments we were dealing with compiled programs, and were able to utilize API's for the (compiled) interpreter of a scripting language (Tcl). In these settings, there is a clear distinction between the machine-code representation of the system under observation and the interpreted language that the dynamic analysis tool is written in.

Java, however, presents an interesting case in that it is already an interpreted language, at least at the bytecode level. One might think that the Java environment, then, does not really benefit from having dynamic analysis tools able to be written in a scripting language. However, we feel that Java is a sufficiently complex language to warrant exploration of making dynamic analyses easier to program.

While Java can access native code resources, and thus could be integrated with external interpreters for scripting languages, there has been enough interest in the combination of Java and scripting languages that open source versions of Java-based interpreters exist for such popular languages as Tcl (Jacl, or Java Tcl) and Python (Jython). This means that we can have the scripted analysis running within the Java environment, which reduces complexity considerably, and future enabling of other probe points should be easier.

To create the interface between the system under observation with the scripting language, we have built a class wrapper generator which uses the Byte Code Engineering Library (BCEL [3]) to generate a wrapper for the public interface methods of a class. In this wrapper we generate calls to the scripting analysis program, with the appropriate data. Thus, the only code needed to be written by the developer interested in some ad-hoc dynamic analysis is the Tcl (Jacl) or Python (Jython) code. As with the C library framework, we offer pre- and post-execution points around the method call, and the interception points are specific rather than generic.

7. Conclusion

It is our hypothesis that developers would more often perform ad-hoc dynamic analyses on the system they are building or maintaining if the cost of creating these ad-hoc analyses was lower than it currently is. To this end, we are experimenting with enabling the analyses to be written in high level scripting languages, with the low-level details hidden and not needing to be the concern of the developers. We have built initial frameworks in CORBA using the Portable Interceptor standard, in C using the dynamic linking phase of library code access, and in Java using the BCEL toolset to access class bytecode. Each of these is centered around method or function call interception, but embody two different styles of access. The CORBA framework enables generic interception points, where all method calls fire the same interception points; while the shared library and Java frameworks enable function/method-specific interception points. We plan to further enhance these frameworks and continue to explore the bounds of usefulness for scripting languages in dynamic analysis.

Acknowledgments

This work was supported in part by the National Science Foundation under grants EIA-9810732 and EIA-0220590. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] M. Auguston, A. Gates, and M. Lujan. Defining a program Behavior Model for Dynamic Analyzers. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, pages 257–262. IEEE Computer Society Press, June 1997.

[2] Interceptors Published Draft with CORBA 2.4+ Core Chapters. Technical Report ptc/2001-03-04, Object Management Group, 2001.

[3] M. Dahm. Byte Code Engineering Library. 2002. http://jakarta.apache.org/bcel/.

[4] S. Eisenbach and C. Sadler. Changing Java Programs. In *Proceedings of the 2001 International Conference on Software Maintenance*, pages 479–487, Nov. 2001.

[5] Gnu C Library. 2002. http://www.gnu.org/.

[6] J. Levine. *Linkers & Loaders*. Morgan Kaufmann, San Diego, CA, 2000.

[7] OMG. The Common Object Request Broker: Architecture and Specification, v2.4.2. Technical Report formal/01-02-01, Object Management Group, 2001.

[8] J. Ousterhout. *Tcl and the Tk Toolkit*. Professional Computing Series. Addison-Wesley, Reading, MA, 1994.