Reverse Engineering Scripting Language Extensions

Daniel L. Moise, Kenny Wong, H. James Hoover
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
{moise, kenw, hoover}@cs.ualberta.ca

Daqing Hou
Avra Software Lab. Inc.
Edmonton, AB, Canada
daqing@cs.ualberta.ca

Abstract

Software systems are often written in more than one programming language. During development, programmers need to understand not only the dependencies among code in a particular language, but dependencies that span languages. In this paper, we focus on the problem of scripting languages (such as Perl) and their extension mechanisms to calling functions with a C interface. Our general approach involves building a fact extractor for each scripting language, by hooking into the language interpreter itself. The produced facts conform to a common schema, and an analyzer is extended to recognize the cross-language dependencies. We present how these statically discovered dependencies can be represented, visualized, and explored in the Eclipse environment.

1. Introduction

It is important to understand software systems written in more than one programming language. For example, a web application may contain a mix of code in Java, HTML, JavaScript, SQL, etc. Legacy systems are typically heterogeneous, with various languages used in their constituent parts. Also, many systems are written with entity, control, and boundary layers, often implemented or generated by different domain-specific languages.

It is not enough to have program understanding tools that consider each language independently as an isolated island. We need to bridge these islands to form a more complete understanding. For example, programmers often need to follow control flows in software, and should not be constrained by language boundaries. It is useful to know if, say, a C function was ultimately called from Perl code, to better assess the impact of potential changes. Also, a more integrated understanding can help in looking for inconsistencies or anomalies, such as malformed or missing stubs in the cross-language mechanism. If a C function is declared to be called from Perl, then a static program analysis can check that the C function indeed exists. Finally, a comprehensive understanding can aid in recovering system architecture [4].

1.1. Multi-Language Code

There are a number of reasons why multi-language systems exist.

- **Efficiency**
  For performance reasons, a high-level language may invoke fragments of code in another lower-level language (e.g., C with embedded assembly). An interpreted language may call functions written in a natively compiled language (e.g., Perl with calls to a C library).
- **Suitability**
  For certain tasks, some languages and notations may be more suitable than others. For example, SQL is the standard notation for manipulating relational data. Scripting languages are useful in gluing together programs. In particular, Perl is very effective at text processing.
- **Reuse**
  Software systems written in different langauges may need to interoperate. Rather than rewriting everything into a single language, the different teams working on each system may continue to use the language they already know.

The space of languages and cross-language interoperability mechanisms is huge. Rather than considering analyses between every pair of languages, it is helpful to divide the space, narrow our focus, and look for general approaches for each partition.

Consequently, interactions between programmatic entities can be broadly categorized as being either loosely coupled or tightly coupled. Loosely coupled interactions may be enabled by sharing a database or file, communicating through network channels, or invoking procedures remotely through the use of middleware. Such interactions typically cross process boundaries. In contrast, tightly coupled interactions happen within a single process, including both transfers of control and exchanges of data.

Tightly coupled cross-language components may interoperate by providing an interface that is invoked using some common calling mechanism (e.g., C convention with arguments pushed onto the stack in reverse order). Similarly, cross-language components may interoperate if each is compiled into a common intermediate language running on a virtual machine interpreter (e.g., Microsoft Common Language Runtime [11], and the Perl 6 Parrot Interpreter [1]). Previously, we have studied the analysis of Java and C code [13] integrated through the Java Native Interface [6].

This paper focuses on scripting languages (e.g., Perl [9, 14], Tcl [18], and Python [15]), and their tightly coupled extension mechanisms to invoking native code (often through a C interface). For performance reasons, the interpreters for these scripting languages are typically written in C (or C++), and thus the cross-language mechanisms tend to be similar. These similarities suggest a more general technique is possible.

1.2. Method

Our method of dealing with a scripting language interacting with C is based on several key steps. First, we reuse a C fact extractor. Second, we need to understand the scripting language and its extension mechanism to calling C code. Third, we write a fact extractor for the scripting language, by hooking into the interpreter implementation itself. Fourth, an analyzer is extended to recognize the cross-language dependencies. Fifth, a visualizer presents the integrated sets of facts for human exploration and understanding.

The extracted facts conform to a common schema. Essentially, a multi-language system can be represented as a set of namespaces, with each containing facts from one language. The schema helps to decouple the cross-language dependency analysis (and downstream tools like visualizers) from the individual language fact extractors.

To illustrate our method, we focus in detail on the Perl to C extension mechanism (Section 2), extraction of Perl facts (Section 3), and analysis and visualization of control dependencies initiated from Perl to C (Section 4). Many Perl modules use this extension mechanism to access deeply into the state of the interpreter (itself written in C) or to call upon C libraries. Our approach is general, since other scripting languages like Tcl and Python work quite similarly. Section 5 highlights further related work, and Section 6 summarizes the paper and outlines directions for future work.

2. Scripting Languages to C Dependencies

This section presents the mechanisms for adding new commands written in C to three widespread scripting languages: Perl [14], Tcl [18], and Python [15]. We use the Perl extension mechanism as the primary example. Details about the extension mechanisms of Tcl and Python are presented in Appendices A and B, respectively.

The core commands of a scripting language can be extended by writing new commands using either the scripting language itself or a system language such as C. There are two main reasons for writing the new commands in C. First, a new command implemented in C is more efficient than the equivalent implemented in the scripting language. Second, and more importantly, for some tasks, it may not be possible to implement the new commands within the scripting language (e.g., accessing a new low-level system device).

2.1. Mechanism for Perl Calling C

Calling C functions from Perl can have several advantages, such as improving the speed of a Perl script by rewriting the time-consuming routines in C, accessing low-level system calls and libraries, or accessing legacy applications that expose a C API. For example, the Perl B module that we studied uses this interoperability mechanism. Consequently, we need to recognize the appearance of such code in a mixed Perl and C system.

To call a C function from Perl, developers need to write the necessary glue code for the Perl interpreter. The glue code usually contains two files: a module file in Perl with the *.pm* extension, and a C file. The Perl module tells the Perl interpreter how to load, dynamically or statically, the library that contains the C function, and the C file puts the C function in the context of the Perl interpreter and associates a new Perl routine with the C function. When we call this Perl routine from a script, the C function associated with the Perl routine will be executed. In Perl's terminology, such a C function is also known as an external subroutine or *XSUB* function.

To understand this process better, consider a simple example. Suppose that we want to create a Perl module *Test* that contains a routine called *test*, and that we want to implement this *test* routine as a C function, instead of a plain Perl routine. The *test.c* file listed in Figure 1 illustrates how the C function (*XS_Test_test*) can be implemented and how the C function can be registered to the Perl interpreter via *boot_Test*.

The *perl.h* header file declares C functions that access the Perl internal data structures, and the *XSUB.h* header file defines a set of macros to write Perl external subroutines. In this example, the C function *XS_Test_test* is the C portion of the glue code. In practice, it is a kind of wrapper which

```
 1. #include "perl.h"
 2. #include "XSUB.h"
 3.
 4. XS(XS_Test_test);
 5. XS(XS_Test_test) {
 6.   dXSARGS;
 7.   if (items != 0)
 8.     Perl_croak(aTHX_ "Usage: test()");
 9.
10.   printf("Test: Perl calls C!\n");
11.
12.   XSRETURN_EMPTY;
13. }
14. XS(boot_Test);
15. XS(boot_Test) {
16.   dXSARGS;
17.   char* file = __FILE__;
18.
19.   XS_VERSION_BOOTCHECK ;
20.   newXS("Test::test", XS_Test_test, file);
21.   XSRETURN_YES;
22. }
```

Figure 1. Listing of *Test.c* file for Perl

typically delegates other C functions to do the real work.

Three macros from *XSUB.h* hide much of the details on how the C function and Perl internals interact:

- **XS** – The *XS* macro defines the standard signature for a new XSUB:

  ```
  #define XS(name) void \
  name(PerlInterpreter *pi, CV *cv)
  ```

  Note that an XSUB function takes two parameters and returns nothing. The first parameter *pi* is a pointer to the current Perl interpreter. The second parameter *CV* is a Perl data structure that represents this function inside the Perl runtime. Other function-specific arguments are made available to an XSUB function implicitly through the Perl runtime stack. Before an XSUB function is invoked, the actual parameters are pushed onto the Perl runtime stack, which can be accessed by the XSUB function through the set of macros defined in *XSUB.h*.
- **dXSARGS** – The *dXSARGS* macro defines the necessary variables for manipulating the Perl stack. For instance, the variable *items* in the example is an integer containing the number of arguments pushed onto the stack by the caller.
- **XSRETURN_EMPTY** – The macro *XSRETURN_EMPTY* indicates that this subroutine does not put anything on the stack as a return value.

The rest of *XS_Test_test* is explained as follows. Line 7 checks if this function has any parameters, and if it does, a usage message is then printed, and the function returns. Note that *Perl_croak* is a Perl internal function which takes two parameters. The first parameter *aTHX_* is a macro that defines a pointer to the current Perl interpreter followed by a comma. The second parameter is the string to be displayed. Line 10 prints a simple message to standard output.

In practice, a Perl module may have a number of such C functions to be called. These C functions comprise a module-specific extension to Perl, and are made known to the interpreter using a specially named registration C function. When a command is issued for Perl to load the module, this registration function is called first. In Figure 2, the function *boot_Test* is the registration function for Perl module *Test*. This function makes the *Test::test* Perl routine known to the Perl interpreter, and associates the C function *XS_Test_test* with the Perl routine *Test::test*. In general, the name of the registration function is formed by prefixing the module name with *boot_*.

The *XS_VERSION_BOOTCHECK* macro checks the module version. The *newXS* macro associates the *XS_Test_test* C function with a Perl subroutine called *test* in a module called *Test* (*Test::test*).

Perl provides two mechanisms for integrating C extensions. One way is to statically link the extension into Perl interpreter code itself. The other way is to dynamically load the extension as a C library. Figure 2 lists the *Test.pm* file that helps to dynamically load the *Test* module. The package statement at line 1 introduces the namespace associated with the module. The package name must match the module name.

```
1. package Test;
2. use strict;
3. use warning;
4.
5. our $version = '1.0';
6. require DynaLoader;
7. bootstrap Test $version;
8. ...
```

Figure 2. Listing of *Test.pm* Perl module file

We can load this module using *use Test;*, and call the Perl routine using *Test::test;*. When the Perl interpreter sees a *use Test;* statement, it searches for a *Test.pm* file to run in all the paths in the predefined *@INC* array. The required *DynaLoader* module is a predefined Perl module that loads shared libraries at runtime (line 6). Perl bootstraps the *Test* module for the given version (line 7). Here, Perl calls its dynamic loader routine, loads the shared library built from the *Test.c* file, and executes the *boot_Test* C function to initialize the *Test* module with the subroutines defined by *newXS*.

	Perl	**Tcl**	**Python**
header files	*perl.h* and *XSUB.h*	*tcl.h*	*Python.h*
declaration	*void (PerlInterpreter *pi, CV *cv)*	*int (ClientData, Tcl_Interp*, int, char*[])* *int (ClientData, Tcl_Interp*, int, Tcl_Obj*[])'*	*PyObject* (PyObject* self, PyObject* args)*
registration	*XS(boot_*packageName*)* *newXS* macro	packageName*_Init* *Tcl_CreateCommand* *Tcl_CreateObjCommand*	*init*ClassName *Py_InitModule* *PyMethodDef* array
loading	*require DynaLoader;* *bootstrap* Test version	*load* Test *package require* Test	*imp.load_dynamic* Test

Table 1. Summary of Perl, Tcl and Python to C extension mechanisms

2.2. Commonalities

The extension mechanisms for scripting languages (such as Perl, Tcl, and Python) are similar, mostly due to the interpreter implementations being written in C. Thus, the basic technique to identify uses of these mechanisms can be generalized with a small language-specific portion. Table 1 summarizes the extension mechanisms, with the slight differences for these languages in four source-level respects. Considered are: the necessary header files to include, the right signatures of C function declarations to use, the registration interface to make C functions known to the interpreter, and the loading commands as scripting language code to enable the extension. Because the extension mechanisms are so similar, code generators like SWIG [16] can assist in generating the many different scripting language dependent wrapers associated with a single C function.

The cross-language dependencies can be identified from the C facts alone (even using a regular expression matching approach with a slight loss of precision). For instance in Perl, one can search for occurrences of the *newXS* macro and associate the new Perl subroutine (e.g., *Test::test*) in the first parameter with the C function in the second parameter (e.g., *XS_Test_test*). Nevertheless, we embarked on a more syntactic and thorough approach to support the exploration of facts from all involved languages (not just C), and to build an infrastructure that would allow for deeper analyses in the future.

3. Fact Extractors

Because of the dynamic nature and often complex semantics of a scripting language, we decided to build the corresponding fact extractor as an extension to the interpreter code itself, to properly reveal the code structure as presented in the internal data structures. We focus here on describing the developed Perl fact extractor. The same technique can be applied for other scripting languages, such as Tcl or Python.

3.1. Perl Extractor

We developed a Perl fact extractor by creating a C extension to the Perl interpreter. Given a Perl script for analysis, we use the interpreter to build up the corresponding intermediate representation in data structures which are then traversed to obtain facts. The extractor begins with the file containing the main body of the program, and recursively examines each used module. An important part of this process is to assign unique identifiers to entities. Next, the extractor reveals all the lexical variables used in each routine. The final step is to analyze the operation subtree of each routine to find the routines being called and the variables being used.

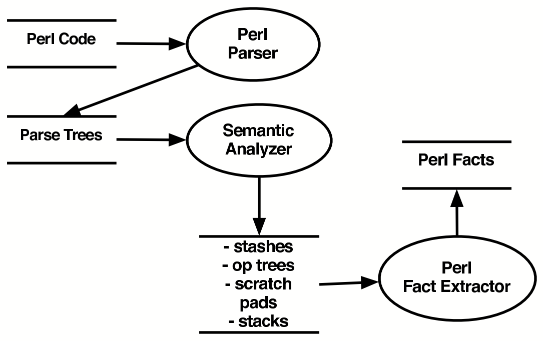

Figure 3. Hooking an extractor into Perl interpreter

Figure 3 depicts the architecture of the Perl interpreter. To understand how the Perl fact extractor works, we focus on the various data structures used.

Name resolution in Perl is non-trivial. Each module has an associated namespace, called a package. Perl assigns a symbol table for each package to store global entities, such as variables and routines, defined by the package. Symbol tables are represented internally as hash tables called *stashes*.

By default each Perl program has a *main* package. A single stash is maintained for *main*. A package can also define sub-packages. In particular, all user-defined, top-level packages are treated as sub-packages of *main*. Therefore, a stash may contain references to other stashes.

In a package, Perl allows the dubious feature of having the same name to refer to different types of entities in the same namespace. Names are resolved through the symbol table, and then refined through a data structure called a typeglob. A typeglob value, called a GV (Glob Value) stores the references to the different entities with the same name.

The GV structure contains references to objects of following basic Perl types: scalar (integer, double, and/or string), array, hash, subroutine, I/O handle, and format name. For example, using this data structure, the entry name *foo* from the stash of a package can represent the scalar *$foo*, the subroutine *&foo* and the array *@foo* at the same time.

Lexical variables (declared with the keyword *my*) are not stored in the stashes. Each subroutine has a data structure called a *scratchpad* to store these variables.

Information about Perl routines are maintained in another data structure called CV (Code Value). CV contains information such as the name of the routine, the scratchpad of the routine, a reference to the stash entry for the routine, the address of the root of its operation tree, and an *XSUB* field. The *XSUB* field is used to distinguish whether the routine is defined externally. For example, if the routine is defined in C, then the field will point to the C function; otherwise, it will be NULL.

Perl defines 351 primitive operations. Statements are translated into operation trees whose nodes are made of these operations. Operation nodes may contain information relevant for fact extraction. For example, each variable reference is represented in some operation node, which maintains an index to the scratchpad through which the variable can be found. In this way, all the variables used in a routine can be obtained.

Facts about routine invocation are extracted by simulating the argument stack. At run-time, Perl uses several stacks, and the most important one is the *argument stack*. This stack stores arguments for invoking the routine plus a data structure that represents the routine to be called. When a routine is invoked, Perl fetches the routine from the stack and executes it.

3.2. Common Schema

Facts to be extracted from each language could be modeled using separate schemas, but this would lead to many similar entities for common notions like namespaces, subroutines, variables, and calls. To address this problem, we have evolved a simpler, common schema that unifies similar notions across the languages of interest [13]. The fact extractors are designed to produce facts that conform to the common schema. This approach also simplifies the implementation of downstream tools, such as visualizers, that use or present the facts.

Node Type	**C/C++**	**Tcl**	**Perl**	**Python**
Class	●	●		●
Comment	●	●	●	●
Constant	●			
Enum	●			
EnumValue	●			
File	●	●	●	●
Function	●	●	●	●
FunctionDecl	●			
GlobalVar	●	●	●	●
Literal	●	●	●	●
LocalVar	●	●	●	●
Macro	●			
MemberVar	●		●	●
Method	●	●	●	●
MethodDecl	●			
Namespace	●	●	●	●
Typedef	●			
Union	●			

Table 2. Schema entities

Table 2 lists the entities in the common schema, and indicates whether each entity is relevant for C/C++, Perl, Tcl, and Python.

4. Implementation

This section describes the implementation and use of our toolset to extract and explore cross-language dependencies (in particular, from Perl to C).

4.1. Toolset Architecture

The toolset consists of portions for fact extraction, analysis, and presentation (see Figure 4). Given a multi-language software system, an independent factbase is produced for each language involved. Each factbase is represented in XML format conforming to a common XML schema, and all factbases can be processed by a common parser. The parser produces an in-memory set of objects that is analyzed to discover cross-language dependencies in the facts. The facts and dependencies can be visualized.

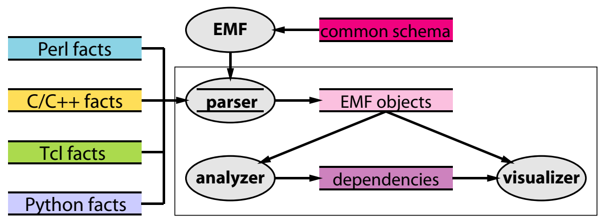

Figure 4. Toolset architecture

To build the toolset, we used the Eclipse platform. In particular, the Eclipse Modeling Framework (EMF) provides services to create and edit data models, as well as a facility to generate code to parse and validate data according to a given schema description. If the data is valid, an in-memory Java object model is constructed automatically. We use EMF to generate the common parser for the XML factbases, using the common XML schema. This approach is especially useful for iterative development. Every time the schema evolves, a new parser can be generated within seconds. Also, the Graphical Editing Framework (GEF) from Eclipse is used to create a rich, graphical editor to present the object model. GEF contains two Eclipse plug-ins: one for graphical drawing and the other for defining an Eclipse workbench window.

4.2. Exploring Cross-language Dependencies

The Perl B module is used as an example to illustrate the exploration of calls in Perl and C code. This module implements the backend of the Perl compiler, which can be used to create opcodes for interpretation. The B module accesses Perl internal data structures through a set of functions defined by the XS mechanism. Other backend utilities, such as cross-reference reports, can be implemented in Perl code, on top of B commands.

Initially, the user specifies the factbases that comprise the system. Figure 5 presents the toolset visualization plug-in, after the Perl and C factbases are loaded. The layout loosely follows Eclipse workbench conventions. From left to right, it contains a navigation view, an editor view for cross-language dependencies (Perl subroutines on-the-left associated to C functions on-the-right), backward and forward call graph views, and an outline view of functions organized by language. Colors are used consistently in the views to distinguish the artifacts of differing languages.

Cross-language dependencies can be used for easing the exploration of control flow from one language to another. In the *Calls From* columnar view, a developer can see what functions call a given one, and follow the calls deeper by exploring more columns to the right. Similarly, the *Calls To* columnar view shows what functions a given one calls.

A developer may want to know what Perl subroutines could be impacted by a change to a C function. For example, changing *XS_B_svref_2object* may influence other Perl routines that use it directly or indirectly, such as *B::C::save_context* and *B::C::save_main*. A developer may also want to know how a Perl external routine is implemented in terms of C functions. For example, as shown in the *Calls To* view, *B::svref_2object* is implemented by a C function *XS_B_svref_2object*, which in turn calls a number of other C functions.

5. Related Work

A number of fact extractors exist for conventional languages, for example, Rigi cparse for C code [20], TkSee with SN [17] as a front-end to extract facts from C/C++ systems [19], Rigi C/C++ Extractor using SN as a front-end to output facts in the Rigi standard format [12], Columbus/CAN for C/C++ source code [3], CppX for C/C++ source code [21], and Chava for Java source code and bytecode (in particular, Java applets) [7]. None of the fact extractors for scripting languages are this mature.

Linos et al. [10] implement a prototype tool called MT (Multi-Language Tool) for understanding multi-language program dependencies. The purpose of MT is to ease the process of detecting, storing, and managing MLDPs (Multi-Language Program Dependencies) found in programs written using a combination of C, C++, and Java. The extractor used in this tool performs a lexical analysis. Our approach is using the parsers from Source Navigator and a precise Perl extractor.

Hassan et al. [5] propose a methodology for maintaining Web applications. A set of extractors is used to analyze the source code of Web applications. The outcome of this analysis is a set of relationships between various components of a Web application. We focus on dependencies from scripting languages and C.

Deruelle et al. [2] describe a method to analyzing distributed multi-language software systems. Several tools help to accomplish this: a multi-language source code analyzer, a software change management module, a profiling tool, and a graphical user interface. The multi-language source code analyzer consists of a set of parsers for each of the languages considered (C, C++, and Java). Each parser is generated using the JavaCC tool based on a language-specific grammar. The source code could also be bytecode, in which case a decompiler is run first. This approach fo-

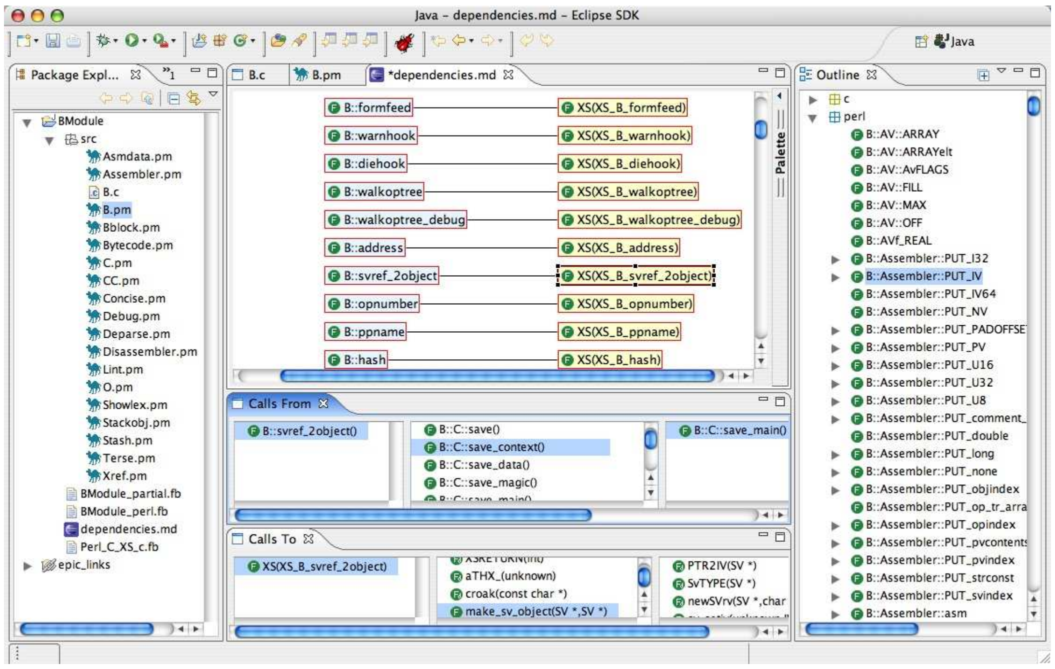

Figure 5. Support Perl-to-C dependencies in Eclipse

cuses more on issues of distributed systems, so it does not extract JNI dependencies between C/C++ and Java code.

Kullbach et al. [8] describe a tool that helps the management of inter-program dependencies for a software application developed in various programming languages, database definitions and job control languages. They use a coarse-grained conceptual model for the individual programming languages, on which an integrated model for the multi-language application is developed. The key observation here is that the inter-program dependencies are defined by job control procedures that coordinate a number of programs and databases.

None of this work focuses on extracting the details of the tight coupling between a scripting language and an extension language (Perl and C in our case).

6. Conclusion and Future Work

Cross language interfaces, and their resulting glue code, are not a large fraction of the code base. But where they sit, and the dependencies they create are important to understanding the system.

Scripting languages pose problems not found in conventional compiled languages, and thus purely lexical approaches do not work effectively. The main problem is that scripting languages make extensive use of dynamic loading, which results in late binding of the details of their module structure. Even though we are hooking into the translation back end, handling late binding is also a weakness of our current approach. We view this work as a first step toward a more dynamic approach to handling scripting languages.

This paper reports our work on extracting and exploiting cross-language dependencies. As a motivating example, we describe the analysis of B module, a case of mixed Perl and C code. We believe that analyses must be integrated seamlessly into the development environments used by programmers today. Consequently, our toolset is built on top of Eclipse. A lesson we learned is that to be flexible and robust, it is necessary to use precise fact extractors.

There are a few directions to proceed with this work. The first is to investigate other kinds of dependencies, such as ones caused by embedding an interpreter in a host language such as C and Java. The second is to evaluate ways in which these cross-language dependencies can be made useful to programmers. Finally, we are also interested in understanding what kind of infrastructure is needed in order to base our analysis directly on an IDE rather than fact extractors. Currently Eclipse provides both JDT (Java Development Tools) and CDT (C++ Development tools). We anticipate that as support for other languages such as Perl, Tcl, and Python is added into the environment, our analysis can then be integrated into Eclipse.

References

[1] Allison Randal and Dan Sugalski and Leopold Tötsch. *Perl*

6 and Parrot Essentials. O'Reilly, 2nd edition, 2004.

[2] L. Deruelle, N. Melab, M. Bouneffa, and H. Basson. Analysis and manipulation of distributed multi-language software code. In *International Workshop on Source Code Analysis and Manipulation*, pages 43–54, 2001.

[3] FrontEndArt Software Ltd., Columbus/CAN. `http://www.frontendart.com`.

[4] A. E. Hassan and R. C. Holt. Architecture Recovery of Web Applications. In *International Conference on Software Engineering*, 2002.

[5] A. E. Hassan and R. C. Holt. A Visual Architectural Approach to Maintaining Web Applications. *Annals of Software Engineering – Special Volume on Software Visualization*, 16, 2003.

[6] Java Native Interface (JNI). `http://java.sun.com/docs/books/tutorial/native1.1`.

[7] J. L. Korn, Y.-F. Chen, and E. Koutsofios. Chava: Reverse Engineering and Tracking of Java Applets. In *Working Conference on Reverse Engineering*, pages 314–325, 1999.

[8] B. Kullbach, A. Winter, P. Dahm, and J. Ebert. Program comprehension in multi-language systems. In *International Workshop on Program Comprehension*, pages 135–143, 1998.

[9] Larry Wall and Tom Christiansen and Jon Orwant. *Programming Perl*. O'Reilly, 3rd edition, 2000.

[10] P. K. Linos, Z. hong Chen, S. Berrier, and B. O'Rourke. A tool for understanding multi-language program dependencies. In *International Workshop on Program Comprehension*, pages 64–72, 2003.

[11] Microsoft Common Language Runtime (CLR). `http://msdn.microsoft.com/netframework/programming/clr`.

[12] D. L. Moise and K. Wong. An Industrial Experience in Reverse Engineering. In *Working Conference on Reverse Engineering*, pages 275–284, 2003.

[13] D. L. Moise and K. Wong. Extracting and Representing Cross-Language Dependencies in Diverse Software Systems. In *Working Conference on Reverse Engineering, 2005*, pages 209–218, 2005.

[14] Perl scripting language. `http://www.perl.org`.

[15] Python scripting language. `http://www.python.org`.

[16] Simplified Wrapper and Interface Generator (SWIG). `http://www.swig.org`.

[17] Source Navigator. `http://sourcenav.sourceforge.net`.

[18] Tcl scripting language. `http://www.tcl.tk`.

[19] University of Ottawa, TkSee. `http://www.site.uottawa.ca/~tcl/kbre`.

[20] University of Victoria, Rigi. `http://www.rigi.csc.uvic.ca`.

[21] University of Waterloo, CPPX. `http://www.swag.uwaterloo.ca/~cppx`.

Appendix

A Mechanism of Tcl Calling C

The Tcl scripting language allows adding new functionality implemented in C. To call a C function from Tcl, developers need to write the glue code necessary to define the function and to register the new command to the Tcl interpreter.

To illustrate this, we provide a simple example that creates two Tcl commands, called *test* and *otest* respectively. These commands are implemented as C functions. Both simply print a string. The two commands show two different ways of defining and registering commands to the Tcl interpreter. The *test.c* file listed in Figure 6 contains the two C functions (*stest* and *otest*), which implement the new commands, and the function (*Test_Init*), which registers the two C functions as Tcl commands.

```
 1. #include "tcl.h"
 2.
 3. int stest(ClientData cd, Tcl_Interp *ti,
 4.       int argc, char *argv[])
 5. {
 6.   printf("Test: Tcl calls C!\n");
 7.   return TCL_OK;
 8. }
 9. int otest(ClientData cd, Tcl_Interp *ti,
10.         int objc, Tcl_Obj *CONST objv[])
11. {
12.   printf("Test: Tcl calls C!\n");
13.   return TCL_OK;
14. }
15. int Test_Init(Tcl_Interp *ti){
16.   Tcl_CreateCommand(ti, "stest",stest,
17.         (ClientData)NULL,
18.         (Tcl_CmdDeleteProc*)NULL);
19.   Tcl_CreateObjCommand(ti, "otest",
20.          otest, (ClientData)NULL,
21.         (Tcl_CmdDeleteProc*)NULL);
22.   Tcl_PkgProvide(ti, "Test", "1.0");
23.   return TCL_OK;
24. }
```

Figure 6. Listing of *Test.c* file for Tcl

The header file *tcl.h* contains the APIs for accessing the Tcl internals. To be registered as a valid Tcl command, a C function must use one of the two signatures as demonstrated by *stest* at lines 3 and 4, and *otest* at lines 9 and 10. The signature of *stest* contains four parameters.

- ClientData– can be used to pass a user-defined data structure to the new command.
- Tcl_Interp– is the interpreter in which the command is executed.
- argc and argv contain the number of parameters and

the array of C string parameters passed to the command, respectively.

Function *otest* also has four parameters, with the first two the same as in the signature of *stest*. The last two parameters are the number of parameter objects and the array of parameter objects passed to this command. Note that in the first case the arguments to the new command are C strings, while in the second case the arguments are Tcl objects. Tcl commands of the second signature have better type-checking support, and may run slightly faster, than the first kind. In the first case C string arguments have to be converted to Tcl objects. Thus in practice the second case is recommended for creating a new Tcl command, and the first case is being maintained only for backward compatibility.

Both *stest* and *otest* print a simple message. They also must return a pre-defined integer value to indicate to the interpreter if an error has occurred during the execution of the command. In our example *TCL_OK* is returned to indicate that there are no errors.

A special C function must be defined to register C functions to Tcl. New Tcl commands may belong to a Tcl package (in our example, the Tcl package is *Test*). The name of this registration function must contain the name of the package followed by *_Init*, and the parameter of this function must be a Tcl interpreter. In our example *Test_init* registers the two new Tcl commands.

Corresponding to the two signatures of C functions for Tcl commands, there are two ways of registering a new Tcl command to the Tcl interpreter: using Tcl APIs *Tcl_CreateCommand* (for C-style string arguments) and *Tcl_CreateObjCommand* (for Tcl objects). These two functions take the same parameters: the interpreter in which the command is executed, the name of the new Tcl command, the pointer to the associated C function that implements the new command, a *ClientData* structure that is given when executing the new command, and *Tcl_CmdDeleteProc*, a pointer to a function that is going to be called when the new command is removed from the interpreter.

The *Tcl_PackageProvide* command at Line 22 declares that a Tcl package named *Test* with version *1.0* is made available to Tcl.

To make the new module available, one needs to compile the *Test.c* file and build a new C library. To use the new Tcl commands, the library must be loaded using either of the Tcl commands *load* or *package require*.

B Mechanism for Python Calling C

The Python scripting language allows adding new functionality to its language implemented in C. The mechanism for building new python modules written in C follows almost the same mechanisms as in the case of Perl and Tcl interpreter. First, following a convention, a C function that is going to integrate with Python is written; this function is often simply wrapper code around some existing C functions. The C function can then be registered to the Python interpreter. Then, a library is built based on the C code, and it is loaded using an existing loading method provided by Python.

A simple example illustrated this mechanism. We create a module called *Test* that contains a Python function *test* implemented in C. The new *test* Python function prints a constant string. The *Test.c* file listed in Figure 7 contains the C implementation of the *test* Python function, and the initialization of the *Test* module with the new function.

```
#include "Python.h"

static PyObject*
test(PyObject *self, PyObject *args) {
  printf("Test: Python calls C!\n");
  Py_INCREF(Py_None);
  return Py_None;
}
static PyMethodDef TestMethods[] = {
 {"test", test, METH_VARARGS, "comment"},
 {NULL, NULL, 0,NULL} /*sentinel*/
};
PyMODINIT_FUNC initTest(){
  Py_InitModule("Test", TestMethods);
}
```

Figure 7. Listing of *Test.c* **file for Python**

The header file *Python.h* contains the Python APIs to access the Python internals. To be registered as a Python command, a C function must possess a signature pre-defined by Python that has two parameters. If the function is meant to be invoked on an object, then the first parameter *self* will be a pointer to the receiver Python object, otherwise it will be *NULL*. The second parameter *args* contains the arguments passed to the Python function.

A Python C function should always return a non-NULL reference to a PyObject. The Python interpreter treats it as an error if a Python C function returns *NULL*. To express the semantics of returning nothing, a function may return a special Python object *Py_None*. However, before returning *Py_None*, the function must increase the reference counter of *Py_None* by calling the *Py_INCREF* macro so that *Py_None* is not to be garbage-collected.

The Python type *PyMethodDef* contains a tuple of four elements that define an entry definition of a Python function. It contains the name of the Python function in the module, the C function that implements the functionality of the new Python function, how to pass the arguments, and a C string comment for the new Python function.

Lines 9–12 defines an array of entry definitions, each of which declares a C function that comprise the *Test* module. In our example, there is only one entry, which associates the Python *test* function with the C function *test*. We use *METH_VARARGS* for passing the Python parameters, which means that Python parameters are passed as a tuple. This is similar to variable length argument lists in C. The tuple can be parsed using the Python API function *PyArg_ParseTuple*.

The function *initTest* creates and initializes the *Test* Python module. This is a special function that informs the Python interpreter of the content of this module; in order for the Python interpreter to recognize it when loading the new module, the name of this registration function must be formed by concatenating *init* and the name of the module. *Py_InitModule* is a Python API function that associates the new Python module *Test* with the array of entry definitions above.

The *Test.c* file is compiled and a new library is built. To make the new Python module available to the Python interpreter, the library is loaded using the *imp.load_dynamic* Python function, which loads a dynamic library containing a Python module (*imp* is a pre-defined Python module and *load_dynamic* is a method of this module). *imp.load_dynamic* searches in the dynamic library for an entry with a name that concatenates *init* and the name of the Python module (in our case *Test*), and executes this C function. This new module can be imported and used by other Python modules using *import Test*.