

Understanding 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, typically 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

There is an important need 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, each implemented or generated by a different suitable language.

It is not enough to have program understanding tools that consider each language independently as an island in isolation. We need to also bridge these islands to form a more complete understanding. For example, programmers often need to follow control flows in software, and this activity should not be constrained by language boundaries. It would be 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 mech-

anism. 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 [3].

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**

A software system may need to interoperate with another one as is, even if written in another language, rather than rewriting everything into a single language. The different teams working on each system may continue to use the language with which they are most familiar.

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 program 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 can be 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 [9]). Previously, we have studied the analysis of Java and C code [10] integrated through the Java Native Interface [5].

This paper, however, focuses on scripting languages (e.g., Perl, Tcl, and Python), and their tightly coupled extension mechanisms to invoking native code (often through a C API). For performance reasons, the interpreters for these scripting languages are typically written in C (and perhaps some C++), and thus, the cross-language mechanisms tend to have similarities. These similarities suggest the potential of a more general technique.

Our method in dealing with a scripting language and 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, typically 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.

One important element is that 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 in the paper, 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 from Perl to C (Section 4). Many Perl modules actually use this extension mechanism to access deeply into the state of the interpreter (itself written in C) or to call upon C libraries. We believe 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 lan-

guages: Perl [12], Tcl [15], and Python [13]. We use the Perl extension mechanism as a primary example. Details about the extension mechanisms of Tcl and Python are presented in Appendix A and Appendix B, respectively. Table 1 provides a summary of the extension mechanisms for the three languages.

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 instead of the scripting language. First, a new command implemented in C is more efficient than the equivalent one implemented in the scripting language. The second reason, and maybe the most important one, is that for some tasks it may not be possible to write the new commands in the scripting language. For example, suppose we want to add a new command to access a new device for which the interface is not available in the scripting language. It is clear that in this case, we are not able to implement the new command using the scripting language.

2.1. Mechanism of Perl Calling C

Perl allows to enhance its language by writing new functionality in C. In this way, Perl can also access parts of a legacy system written in C.

Calling C functions from Perl can have several advantages, for example, improving the speed of a Perl script by rewriting the time-consuming routines in C, accessing low-level system calls and libraries, or accessing other applications that expose 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/C system.

To call a C function from Perl, developers need to write the necessary glue code for the Perl interpreter to understand the C code. 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 the new Perl routine with the new C function. When we call the 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 *XSUB*.

To understand better this process, consider a simple example. Suppose that we want to create a Perl package *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 internal via

boot_Test.

```
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

The two standard header files contain the APIs for accessing the Perl internal and for writing external subroutines, respectively. The *perl.h* header file contains the API functions to access the Perl internal data structures, and the *XSUB.h* header file defines a set of macros to write Perl XSUBs.

In this example, the C function *XS_Test_test* is the glue code. To keep it simple, all that this function does is to print a string. In practice, one can imagine to call other C functions to make it more powerful. It uses three macros from *XSUB.h*, which hide much of the details on how the C function and the Perl internal 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 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. It is provided as some C XSUB functions may need to access *CV*.

These two are standard parameters from the Perl internal. Other function-specific arguments are made available to the XSUB functions implicitly through the Perl runtime stack. Before an XSUB is invoked,

its arguments are pushed onto the Perl runtime stack, which can be accessed by the XSUB 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* appearing in the example is in fact an integer, which contains the number of arguments pushed on 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 API that 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 achieves the functionality of the function: output a simple message to the standard output.

Each C extension package must have one C function to register its C functions to the Perl internal. When a command is issued to the Perl interpreter to load such a package, this registration function will actually be invoked behind the scene. In Figure 2, the function *boot_Test* is the registration function for package *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*. Note that the choice of the name *boot_Test* is not incidental: the name of a registration function is formed by prefixing the package name *Test* with *boot_*.

The *XS_VERSION_BOOTCHECK* macro requires a checking of the version of this package. The *newXS* macro is doing the work of hooking up the *XS_Test_test* C function as a Perl subroutine called *test* in a package called *Test* (*Test::test*).

Perl provides two mechanisms for integrating C extension packages. One way to add a C extension package to the Perl internal is to recompile Perl. The other way is to dynamically load the C library. Figure 2 lists the *Test.pm* file that helps to dynamically load the *Test* module.

```
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

	Perl	Tcl	Python
header files	<i>perl.h</i> and <i>XSUB.h</i>	<i>tcl.h</i>	<i>Python.h</i>
declaration	<i>void (PerlInterpreter *pi, CV *cv)</i>	<i>int (ClientData, Tcl_Interp*, int, char*[])</i> <i>int (ClientData, Tcl_Interp*, int, Tcl_Obj*[])</i>	<i>PyObject* (PyObject* self, PyObject* args)</i>
registration	<i>XS(boot_packageName)</i> <i>newXS</i> macro	<i>packageName_Init</i> <i>Tcl_CreateCommand</i> <i>Tcl_CreateObjCommand</i>	<i>initClassName</i> <i>Py_InitModule</i> <i>PyMethodDef</i> array
loading	<i>require DynaLoader;</i> <i>bootstrap</i> Test version	<i>load</i> Test <i>package require</i> Test	<i>imp.load_dynamic</i> Test

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

We can load our 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 in all the paths defined in the predefined `@INC` array. The `DynaLoader` module is a predefined Perl module that provides routines to link shared libraries at runtime. Line 7 in the `Test.pm` file tells Perl to bootstrap the `Test` module with the given version. Perl calls its dynamic loader routine, loads the shared library built from the `Test.c` file, and executes the `boot_Test` C function for initializing the `Test` Perl package with the subroutines defined by `newXS`.

2.2. Discussion

Table 1 summarizes the mechanisms through which Perl, Tcl and Python can be extended using C. Four common aspects for extension are considered: the necessary header files to include, declaration of C functions, registration, and loading. The declaration aspect concerns the signatures that the C functions should have. Registration is about the API that register the C functions to the interpreter. The loading aspect is about mechanisms that enable the scripting language to use the new functionality.

Dependencies can be identified from the C facts. For example, for Perl we associate the new Perl subroutine found in the first parameter of the `newXS` macro with the second one, which is a pointer to the C function. In the example from Figure 1, the `Test::test` new Perl subroutine is associated with the `XS_Test_test` C function.

We note two points here. One, although the dependencies can be identified from C facts alone, facts from scripting languages are still needed in order to be able to navigate across languages. Two, three scripting languages follow a conceptually similar approach in defining extensions. Despite of this conceptual similarity, each scripting language uses a slightly different subset of C constructs to define extensions. A precise C fact extractor that can reliably provide all relevant facts is needed in order to support all three

scripting languages.

3. Fact Extractors

Currently our toolset includes fact extractors for C, C++, Java, and Perl. For C, C++, and Java, we are using Source Navigator [14] to obtain facts. The facts are stored in an internal database. Source Navigator also provides an API for manipulating the internal database. We use this API to produce factbases that comply to the common schema. We enhance the factbase produced by Source Navigator parsers by providing the actual arguments for function calls, which is needed for identifying cross-language dependencies.

Because of the dynamic nature of the scripting languages, we decided to build the scripting language extractors inside the source code of the interpreter to reveal the facts accurately from their internal structures. We applied this technique successfully for developing a Perl 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 modifying the Perl interpreter. Given Perl code, we use the interpreter to build up corresponding data structures which are then traversed to extract facts. Starting from the main package, the extractor interrogates its stash to extract out the content of the package. It also retrieves all the used packages and repeats the above step for each used package. The goal of this step is to assign unique identifiers to entities. Next, the extractor provides 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 depicts the architecture of the Perl interpreter. The Perl interpreter has a standard compiler structure. In the following, we will focus on the various data structures

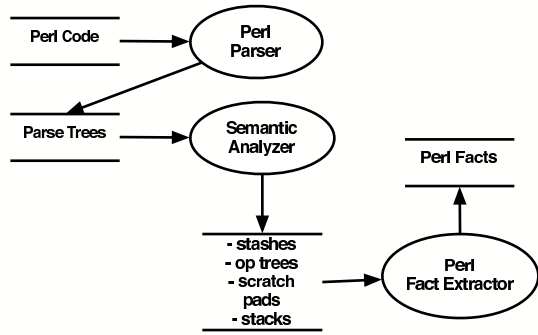


Figure 3. Hooking an Extractor into Perl Architecture

used, to understand how the Perl fact extractor works. The Perl interpreter is written in C.

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 hashes, which are also 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 same name to refer to different entities. A data structure called GV (Glob Value) is assigned to the name to store references to the entities.

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 extracted from source code are represented in an entity-relationship schema. In the schema, entities are things like namespaces, functions, and variables. Relationships include calls, defines, and uses. We evolved a common schema suitable for our languages of interest [10]. A unifying schema reduces the number of concepts, and thus simplifies the implementation of other tools that depend on the schema, such as graph visualizers.

This common schema can be seen as a common schema for the individual languages, enhanced with elements that are particular for an individual language.

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

Table 2. Schema Node Types

To simplify the schema, some elements from the schema represent similar things across different languages. The node types and their meaning for some languages is illustrated in Table 2. The first column consists of all the node

types of our schema, and the next column represents the meaning of the node type with respect to various languages. For example, the *Function* node type refers to a function in C/C++, to a procedure in Tcl, or a subroutine in Perl. An empty cell means the node type does not have a corresponding construct in the language. For example, the *Enum* node type does not exist for the Tcl language.

4. Implementation

We have implemented, within the Eclipse platform, a toolset for extracting dependencies from Perl to C. Our toolset also includes applications of the extracted facts. This section presents the implementation of the toolset. To illustrate Perl to C dependencies and their applications, we analyzed the Perl compiler, B module, an example of Perl routines calling C functions.

4.1. Toolset Architecture

Our toolset adopts the standard reference architecture for reverse engineering tools. It consists of parts for facts extraction, analysis, and presentation. Given a multi-language software system, the toolset first extracts facts out of each language. Facts from each language form an independent factbase. Therefore if the system uses n languages, there will be n initial factbases created. In the next phase, dependencies between each pair of languages are derived. Finally, all the facts are used for query and visualization.

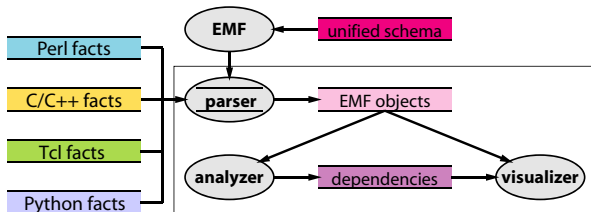


Figure 4. Toolset architecture

4.1.1 Eclipse Modeling Framework (EMF)

Eclipse Modeling Framework (EMF) is both a modeling framework and code generation facility for managing structured data. (EMF supports a subset of UML called Ecore.) As a modelling framework, it provides an API for creating and editing data models. As a code generation facility, it generates code for a given data model specification. The generated code can then be used to parse and validate instances of the data model. If the instance is valid, an in-memory Java object model is built automatically. In this respect, EMF is similar to a compiler generator.

We use EMF to generate a factbase parser. EMF can accept a number of data model specifications in different formats, including XML schema and annotated Java. Our unified factbase schema is written as an XML schema. EMF takes the XML schema as input and generates a factbases parser. The parser is then used to read in factbases and produces the in-memory factbases (object models) for analysis.

EMF is a useful technique for iterative development. It has taken us a number of iterations before settling down to a stable schema. With EMF, every time the schema is changed, a new parser can be easily generated within seconds. This has sped up our development considerably.

4.1.2 Dependency Analysis

The dependency analysis component contains a set of algorithms that calculate cross-language dependencies. For each pair of languages, say C and Perl, there would be two algorithms, one for C-to-Perl dependencies and one for Perl-to-C dependencies.

4.1.3 Graphical Editing Framework (GEF)

The Graphical Editing Framework (GEF) allows developers to take an existing object model and quickly create a rich graphical editor. GEF contains two Eclipse plug-ins: one for graphical drawing and the other for defining an Eclipse workbench window. The user interface of our toolset is developed with GEF.

4.2. Applying Cross-language Dependencies

In this section, we use the Perl module B as an example to demonstrate an application of the extracted dependencies to bridge the call graphs from two languages.

The Perl module B comprises Perl and C code. B implements a Perl compiler that compiles a Perl program into an executable instead of interpreting it. The compilation is done by accessing the data structures inside the Perl interpreter. B accesses Perl internals through a set of APIs defined by the XS mechanism. Other backend tools, such as cross referencing, can then be implemented in Perl on top of B.

To extract cross-language dependencies and demonstrate their application to exploring cross-language call graphs, we built an Eclipse plug-in, exploiting two Eclipse frameworks: EMF and GEF. Figure 5 is a screenshot of the plug-in after two factbases of Perl and C are loaded. The layout of the screen follows that of a standard Eclipse workbench page. From left to right, it contains a navigation view, an editor, and an outline view. The two views at the bottom are for forward and backward exploration of a call graph.

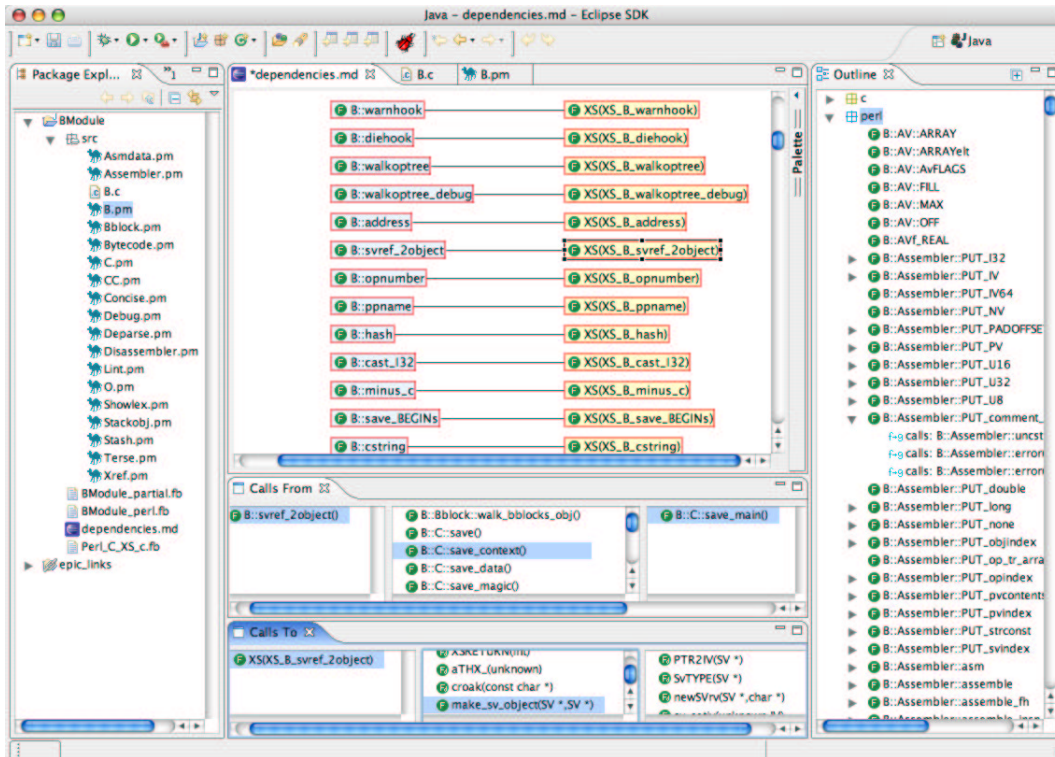


Figure 5. Support Perl-to-C dependencies in Eclipse

There are two steps in using the plug-in. First, the user specifies to the plug-in all the factbases that comprise the system. The plug-in loads in all the factbases and shows the facts in the outline view. Second, the user may instruct the plug-in to perform cross-language analysis and the result is shown in the editor view. Users can also manually edit the resulting diagrams.

The outline view visualizes the facts in a tree hierarchy. For each language, there is a top-level node representing a namespace that contains all the facts for that language. Figure 5 shows two top-level nodes, for Perl and C, respectively. At the next level are functions. In particular, the call relationship between a caller and its callees is displayed as a tree (Figure 5). Colors are used to distinguish languages. In the picture, blue is used for Perl and orange for C.

The editor is used to display and edit a dependency diagram. The user can instruct the plug-in to analyze the dependencies from code in one language to another. For the B module example, pairs of nodes are displayed in the editor with a line connecting them. The node at the left-hand side represents a Perl routine, and the right one represents a C function. They are colored in the same color as their corresponding top-level nodes from the outline view. Thus it becomes apparent which language a node belongs to. Note that in this case the relation is one-to-one and therefore we

can simply connect the two nodes by a line. More complex relations would require other ways of visualization.

While a dependency is a useful piece of information in itself, it can be used for other purposes such as bridging two languages. We demonstrate this by providing both forward and backward call graph exploration in our toolset. With forward call graph exploration, a programmer examines the functions called by a selected function (the *Calls To* view in Figure 5). In backward exploration, a programmer examines the functions that call a selected function (the *Calls From* view).

Such exploration can be useful in a number scenarios. For example, when a C function is changed, one may want to know what Perl routines depend on it. 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 programmer 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 *XSXS_B_svref_2object*, which in turn calls a number of other C functions.

5. Related Work

A number of fact extractors exist, for example, Rigi cparse for C code [17], TkSee with SN [14] as a front-end to extract facts from C/C++ systems [16], Rigi C/C++ Extractor using SN as a front-end to output facts in the Rigi standard format [11], Columbus/CAN for C/C++ source code [2], CppX for C/C++ source code [18], and Chava for Java source code and bytecode (in particular, Java applets) [6].

Linos et al. [8] implemented 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 languages. The extractor used in this tool is based on a lexical analysis. Our approach is syntactic, using extractors from Source Navigator and a precise Perl extractor.

Hassan et al. [4] proposed a methodology for maintaining their 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. [1] described 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 focuses more on issues of distributed systems, so it does not extract JNI dependencies between C/C++ and Java code.

Kullbach et al. [7] described 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. Their approach uses 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.

6. Conclusion and Future Work

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] 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.
- [2] FrontEndArt Software Ltd., Columbus/CAN. <http://www.frontendart.com>.
- [3] A. E. Hassan and R. C. Holt. Architecture Recovery of Web Applications. In *International Conference on Software Engineering*, 2002.
- [4] 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.
- [5] Java Native Interface (JNI). <http://java.sun.com/docs/books/tutorial/native/1.1>.
- [6] J. L. Korn, Y.-F. Chen, and E. Koutsoufios. Chava: Reverse Engineering and Tracking of Java Applets. In *Working Conference on Reverse Engineering*, pages 314–325, 1999.
- [7] 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.
- [8] 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.
- [9] Microsoft Common Language Runtime (CLR). <http://msdn.microsoft.com/netframework/programming/clr>.
- [10] D. L. Moise and K. Wong. Extracting and Representing Cross-Language Dependencies in Diverse Software Systems. In *Working Conference on Reverse Engineering, 2005*.
- [11] D. L. Moise and K. Wong. An Industrial Experience in Reverse Engineering. In *Working Conference on Reverse Engineering*, pages 275–284, 2003.
- [12] Perl scripting language. <http://www.perl.org>.
- [13] Python scripting language. <http://www.python.org>.

- [14] Source Navigator. <http://sourcenv.sourceforge.net>.
- [15] Tcl scripting language. <http://www.tcl.tk>.
- [16] University of Ottawa, TkSee. <http://www.site.uottawa.ca/~tcl/kbre>.
- [17] University of Victoria, Rigi. <http://www.rigi.csc.uvic.ca>.
- [18] University of Waterloo, CPPX. <http://www.swag.uwaterloo.ca/~cppx>.

Appendix

A Mechanism of Tcl Calling C

The Tcl scripting language allows to add new functionality implemented in C to its language.

To call a C function from Tcl, developers need to write the glue code necessary to register the new command to the Tcl interpreter. The glue code contains the definition of the new command, and the registration of the new command to the interpreter. The registration of the new command performs also the linkage between the Tcl command and the C function associated with the Tcl command. Therefore, when we call the Tcl command from a script, the C function associated with the Tcl command will be executed.

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.

The header file *tcl.h* contains the APIs for accessing the Tcl internal.

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.

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

```

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

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. Note that both functions must return a pre-defined integer value to indicate if an error has occurred during the execution of the command. In our example *TCL_OK* is returned to inform the Tcl interpreter 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* or *Tcl_CreateObjCommand*. *Tcl_CreateCommand* is used to register functions that assume all their arguments are C-style strings, and *Tcl_CreateObjCommand* is used to register those whose ar-

guments are 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 *load* or *package require*, both are Tcl commands.

B Mechanism of Python Calling C

Python scripting language allows to add 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.

We provide a simple example illustrating 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.

The header file *Python.h* contains the Python APIs to access the Python internal.

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 semantic 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.

```

1. #include "Python.h"
2.
3. static PyObject*
4. test(PyObject *self, PyObject *args) {
5.     printf("Test: Python calls C!\n");
6.     Py_INCREF(Py_None);
7.     return Py_None;
8. }
9. static PyMethodDef TestMethods[] = {
10.  {"test", test, METH_VARARGS, "comment"},
11.  {NULL, NULL, 0, NULL} /*sentinel*/
12. };
13. PyMODINIT_FUNC inittest(){
14.     Py_InitModule("Test", TestMethods);
15. }

```

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

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 the variable arguments 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 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 Python functions defined before.

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 (*im* 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*.