OpenRefactory/C: An Infrastructure for Building Correct and Complex C Transformations

Munawar Hafiz, Jeffrey Overbey, Farnaz Behrang, and Jillian Hall
Auburn University
{munawar,joverbey,fzb0012,jnh0008}@auburn.edu

Abstract

OpenRefactory/C is a refactoring tool and, more generally, an infrastructure that resolves the challenges of building C program transformations. In this paper, we describe its architecture, extensibility features, and the transformations implemented. We also discuss features that will make OpenRefactory/C attractive to researchers interested in collaborating to build new C program analyses and transformations.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement

Keywords Program Transformation, C

1. Introduction

C is one of the most popular programming languages. Yet, when it comes to making even a small change to a program, such as renaming a variable, C programmers do it manually. IDEs for C support a few refactorings, but these tend to be slow and buggy [5]. There are two challenges that make it hard to build tools for transforming C programs. First, C programming IDEs ignore multiple configurations of C preprocessor because of the complexity. The resulting transformations are inaccurate. Second, IDEs for C do not support sophisticated static analyses, e.g., Eclipse CDT only supports name binding, type analysis and limited control flow analysis. Without data flow analysis, it is impossible to implement any non-trivial transformations. Preprocessors and multiple configurations make static analysis even more complicated.

OpenRefactory/C [7] is an infrastructure for building refactorings and similar source-level transformations for C, while correctly handling all of the complexities of C. Some of these goals are still a year or two away, but work is progressing rapidly. The OpenRefactory/C codebase consists of more than 400,000 lines of Java code. It contains support for a number of static analyses, including data flow and alias analysis, and work is currently under way to support the analysis and transformation of programs containing preprocessor directives. We have developed 12 transformations on this platform. Some of these are commonly used refactorings; others are behavior-enhancing transformations aimed at fixing security vulnerabilities [1, 6].

This paper makes three contributions:

- It describes OpenRefactory/C's architecture (Section 2).
- It explains how OpenRefactory/C can be used from a variety of text editors and IDEs (Sections 3 and 4).
- It discusses the refactorings and program transformations implemented and our testing methods (Section 5.3).

2. OpenRefactory/C Architecture

OpenRefactory consists of multiple front ends communicating with a back end following a JSON protocol (Figure 1). *Front ends* provide the user interface. The *back end*, written in Java, contains the infrastructure to access the file system, parse and analyze source code, and create source code patches. The back end supports many programming languages; OpenRefactory/C is the C-specific infrastructure. The following sections describe these.

3. Front Ends

Developers write C code in many environments, from text editors like Vim to full-fledged IDEs like Eclipse. Anecdotally, we have found that developers that prefer Vim, for ex-

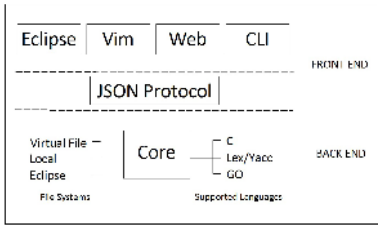

Figure 1. OpenRefactory Architecture

WRT '13, October 27, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2541348.2541349

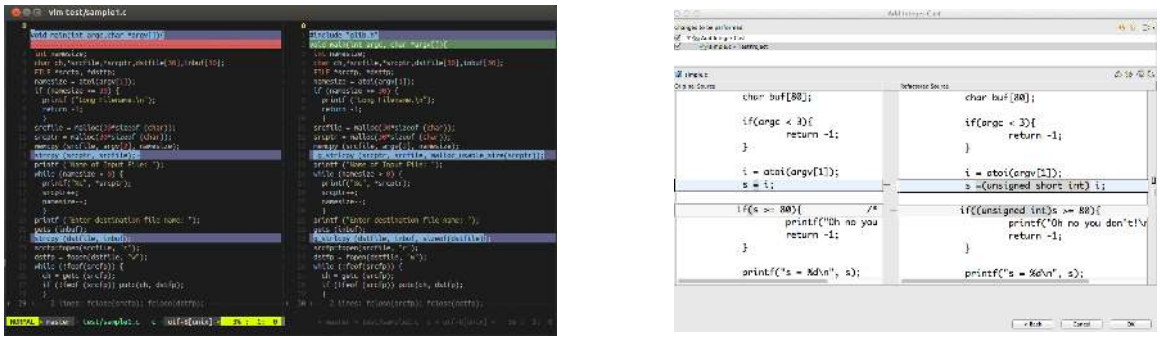

Figure 2. Program Transformations in Diff-view in Vim and Eclipse

ample, will not switch to Eclipse just for refactoring support. So, OpenRefactory/C was designed to be accessible from all of these environments. Currently, OpenRefactory/C supports four front ends: an OpenRefactory Eclipse plug-in, a Vim plug-in, a command line user interface, and a Web demo at the `www.openrefactory.org` Web site. Figure 2 shows refactorings in Vim and Eclipse. We are working on integrating support for Notepad++ and Sublime Text. The Eclipse plug-in contains additional visualizations for various static analyses to assist refactoring tool builders; Figure 3 shows a visualization of how a variable's declaration is bound to its references.

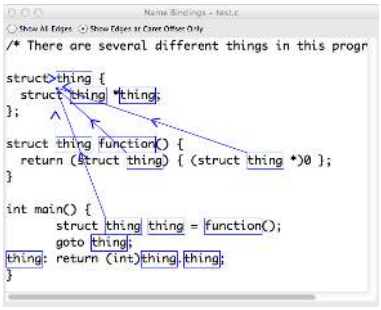

Figure 3. Name Binding Visualization in Eclipse

4. JSON Protocol

OpenRefactory/C's four front ends are all written in different programming languages—Eclipse in Java, Vim in Vim Script, etc. This poses an interoperability problem, since these front ends must communicate with the same back end (written in Java).

Shanbhag [16] addressed a similar problem building an interface for Garrido's CRefactory [2]: the refactoring engine, written in Smalltalk, needed to be run from the Emacs text editor. But Emacs plug-ins are written in Emacs Lisp. This was resolved by having the refactoring engine act as a server, with the client communicating via sockets. Results were sent to the client as patch files; Emacs could then display differences to the user and patch the relevant files.

Our approach is similar. However, unlike CRefactory, the back end does not act as a TCP/IP server. Instead, the front end is responsible for launching the back end process, and the two communicate via standard I/O, i.e., via pipes. (In the case of the Web demonstration, the system is distributed: the front end is written in JavaScript and runs in a Web browser, while the back end runs on the Web server.)

Communication between the front and back ends is formatted in JavaScript Object Notation (JSON). The client sends a query (formatted as JSON) to the server, which then interprets the command, performs the requested action, and returns a reply (again formatted as JSON). JSON was chosen because it is human-readable (unlike a binary protocol), concise, and easy to parse; JSON parsers exist for nearly every language, including Vim Script. A complete specification of the JSON protocol is available [12].

A sample dialogue between the Web front end and the OpenRefactory back end is shown in Figure 4. The client opens the connection and sets the protocol version. Since the Web front end does not share a file system with the back end, it issues a *put* command to transmit the contents of *file.c* to the back end. (The Eclipse and Vim front ends would simply point the back end to the project directory.) Then, the front end issues the *params* command to determine what input is required. Finally, it issues the *xrun* command to perform the transformation and retrieve the changed file. The *close* command shuts down the refactoring engine.

5. Back End

5.1 Back End Architecture

The OpenRefactory back end consists of a language-agnostic core and two types of plug-ins:

- *Language-agnostic core.* The OpenRefactory core provides an abstraction of the file system and provides interfaces that all transformations must implement. It also provides the implementation of the JSON server.
- *File system plug-ins.* Plug-ins add support for file systems. One plug-in supports the local file system, used by the Vim and command-line user interfaces; another provides a "virtual" file system to support the Web demo (for security and performance reasons); a third supports the Eclipse File System.

```
C: {"command":"open", "version":0.1}
S: {"reply":"OK"}
C: {"command":"setdir","mode":"web"}
S: {"reply":"OK"}
C: {"command":"put","filename":"file.c","contents":"int main() {\n return 0;\n}\n"}
S: {"reply":"OK"}
C: {"command":"params","transformation":"rename","textselection":{"filename":"file.c","offset":4,
   "length":0}}
S: {"reply":"OK","params":[{"default":"","label":"New name:","prompt":"Please enter a new name for this
   identifier.","type":"string"},{"default":false,"label":"Replace occurrences in comments","prompt":"Should
   occurrences in comments be replaced as well?","type":"boolean"}]}
C: {"command":"xrun","transformation":"rename","textselection":{"filename":"file.c","offset":4,
   "length":0},"arguments":["abcde",false]}
S: {"reply":"OK","description":"Rename main to abcde",
   "files":[{"filename":"file.c","contents":"int abcde() {\n return 0;\n}\n"}],
   "log":[{"message":"The program entrypoint must be called \"main\". Renaming the main function will cause the
   program to have no entrypoint.","severity":"error"}]}
C: {"command":"close"}
S: Server terminates
```

Figure 4. Sample dialogue between the OpenRefactory Web demonstration front-end/client (**C**) and back-end/server (**S**).

- *Language-specific plug-ins.* Plug-ins add support for individual languages to the language-agnostic core. Most of our current work has focused on supporting C, but plug-ins for Lex/Yacc and several other languages are in the early stages of development.

5.2 C Infrastructure

5.2.1 Syntax Analysis and Source Manipulation

OpenRefactory/C uses a custom parser that supports the ISO C99 standard [9] with GNU extensions. It is integrated with a preprocessor that was hand-written for OpenRefactory/C.

The preprocessor currently supports single-configuration preprocessing (i.e., only one branch of an `#ifdef` directive is analyzed); however, it is currently being extended to support multiple configurations following techniques described by Overbey et al. [13] and Gazzillo et al. [4]. The original work on refactoring in the presence of multiple configurations is due to Garrido [2, 3].

The decision to construct a custom parser and preprocessor for OpenRefactory/C was a difficult one, given the number of parsers already available. The motivation was twofold.

First, one of the main research goals for OpenRefactory/C is to support analysis and transformation of code containing C preprocessor directives, with correct results under all feasible preprocessor configurations. These changes require intricate modifications to every part of the infrastructure: the preprocessor, the lexical analyzer, the parser, the abstract syntax tree, every static analysis, and every transformation. In OpenRefactory/C, these components were all built, from the beginning, with this agenda in mind. (For example, the parsing algorithm we intend to use to support multiple configurations requires an LALR(1) parser [4, 13], and analyzing name bindings under multiple configuration requires augmenting name bindings with information about the preprocessor configurations under which that name binding is valid [2].) After careful consideration, we determined that building an infrastructure from scratch was the most cost- and time-effective way to guarantee that all of our technical constraints could be met.

Building a custom infrastructure also allowed us to have confidence that we could provide a satisfactory API for third-party developers to extend OpenRefactory/C with new refactorings. OpenRefactory/C's C parser and rewritable abstract syntax tree are generated using Ludwig [13], which was also used to generate the syntactic manipulation infrastructure in Photran [14], a refactoring tool for Fortran. Photran 8.1 has 39 refactorings contributed by over 40 different developers. Since Photran's source manipulation API has been used successfully by a wide audience—and it has had several years to mature and stabilize—OpenRefactory/C adopted a similar API.

Internally, OpenRefactory/C uses mutable abstract syntax trees (ASTs) as its primary program representation. Source code manipulation is performed by modifying the AST and later traversing the tree to output the revised source code (or create a patch file). All AST nodes implement a common interface that includes methods to perform tree traversals using the Visitor pattern, find nodes by type, determine preprocessor constructs affixed to nodes, and of course, manipulate the source code associated with a node.

5.2.2 Static Analysis for C

OpenRefactory/C supports several static analyses:

- *Name Binding Analysis.* OpenRefactory/C stores name binding information as relationships among nodes in the AST. Any AST node representing a name reference can be queried to return the corresponding declaration node. The name binding edges are used in some checks for behavior preservation [11, 15].
- *Type Analysis.* Types are dynamically computed as queries to the AST; this, in turn, requires name binding information.

- *Control Flow Analysis.* Control flow predecessors and successors are determined by querying AST nodes; they are computed dynamically using an improved version of Morgenthaler's [10] algorithm.
- *Alias Analysis.* OpenRefactory/C supports an inclusion-based (Andersen-style) alias analysis for local variables. Constraints are derived from the source code and solved using Hardekopf's [8] algorithm.
- *Data Flow Analysis.* OpenRefactory/C stores intraprocedural reaching definitions and definition-use relationships in the AST.
- *Dependence Analysis.* OpenRefactory/C supports scalar data dependence analysis for local variables, as well as computation of control dependences. Control and data dependences check behavior preservation of transformations that move code within a procedure.

5.3 Transformations

Currently 12 transformations have been implemented in OpenRefactory/C infrastructure. Of them, 7 are refactorings and 5 are behavior enhancing program transformations.

- *Refactorings.* OpenRefactory/C supports simple refactorings, such as RENAME, ADD FUNCTION DECLARATION, MOVE EXTERNAL DECLARATION, CORRECT INDENTATION, and MOVE STATEMENT. It also supports composite refactorings, such as EXTRACT FUNCTION (composed of 11 micro-refactorings) and EXTRACT LOCAL VARIABLE (composed of 7 micro-refactorings).
- *Behavior Enhancing Program Transformations.* ADD INTEGER CAST, CHANGE INTEGER TYPE, REPLACE ARITHMETIC OPERATOR, SAFE LIBRARY REPLACEMENT, and SAFE TYPE REPLACEMENT are security-oriented program transformations [1, 6]. They are complex transformations that make extensive use of OpenRefactory/C's static analyses; they are intended to improve the security of systems by removing "bad-path" behaviors (like buffer overflows) while otherwise preserving behavior.

Previously, we presented a simple yet extremely effective approach [5] to detect unique, real bugs in refactoring engines and to estimate their reliability. Through our research we concluded that refactoring engines are underused mostly due to unpredictability of the refactorings, and that there is still a need for improvement and development of new infrastructure for building more reliable refactoring engines. We used this approach to test the program transformations in OpenRefactory/C by applying the refactorings on a large number of places in real software projects and effectively clustering the failures to find bugs. EXTRACT FUNCTION REFACTORING has been tested using this approach. We applied each micro-refactoring on more than 12000 targets in three software projects and found and fixed 42 bugs.

6. Future Work

Work is underway to allow OpenRefactory/C to support multiple preprocessor configurations. This will allow it to transform un-preprocessed source code, exactly as the programmer sees it, and guarantee correctness under all feasible macro configurations. Additionally, it will support transformations for mixed-language programs.

An open source release of OpenRefactory/C is planned for the near future. We welcome contributions and collaborations with other researchers. Correct analysis and transformation of C source code is a vast and difficult topic, and we believe OpenRefactory/C is well positioned to be the platform researchers will use to address it.

Acknowledgments

This work is supported by the National Science Foundation under Grant Nos. 1217271 and 1217271-REU.

References

[1] Z. Coker and M. Hafiz. Program transformations to fix C integers. In *ICSE*, pages 792–801, 2013.

[2] A. Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, UIUC, 2005.

[3] A. Garrido and R. Johnson. Refactoring C with conditional compilation. In *ASE*, 2003.

[4] P. Gazzillo and R. Grimm. SuperC: Parsing all of C by taming the preprocessor. In *PLDI*, pages 323–334, 2012.

[5] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov. Systematic testing of refactoring engines on real software projects. In *ECOOP*, Volume 7920 of *LNCS*, pages 629–653, 2013.

[6] M. Hafiz. *Security On Demand*. PhD thesis, UIUC, 2010.

[7] M. Hafiz and J. Overbey. OpenRefactory/C: An infrastructure for developing program transformations for C programs. In *OOPSLA*, 2012.

[8] B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299, 2007.

[9] International Organization for Standardization. *ISO/IEC 9899:TC3: Programming Languages — C*. Sep 2007.

[10] J. D. Morgenthaler. *Static Analysis for a Software Transformation Tool*. PhD thesis, UCSD, 1997.

[11] J. Overbey and R. Johnson. Differential precondition checking: A lightweight, reusable analysis for refactoring tools. In *ASE*, pages 303–312, 2011.

[12] J. Overbey and A. Lewis. OpenRefactory protocol specification. `http://www.openrefactory.org/doc/dev/protocol.pdf`, 2013.

[13] J. L. Overbey and R. E. Johnson. Generating rewritable abstract syntax trees. In *SLE*, Volume 5452 of *LNCS*, pages 114–133, 2008.

[14] Photran – An Integrated Development Environment and Refactoring Tool for Fortran. `http://www.eclipse.org/photran/`.

[15] M. Schäfer, T. Ekman, and O. de Moor. Sound and extensible renaming for Java. In *OOPSLA*, pages 277–294, 2008.

[16] C. K. Shanbhag. *The Design of a User Interface for a Refactoring Tool for C*. Master's thesis, UIUC, 2003.