Program Manipulation via Interactive Transformations

Extended Abstract

Marat Boshernitsan (maratb@cs.berkeley.edu)

Research Area

Interactive language-aware software engineering tools.

Introduction

Today almost all software is created and maintained using interactive tools on a workstation with a bit-mapped color display and respectable computing power. However, despite considerable research on methods to improve software quality and developer productivity, most working developers use software tools similar to those available decades earlier. The usual form of input is typed text; the most common forms of browsing and manipulation of software artifacts are reading text, typing text, and doing text-based searches and substitutions. The use of text-based interaction with minimal syntax knowledge and very little structural or high-level language awareness has limitations. On the one hand, developers utilize high-level linguistic structure and programming language semantics when thinking about and discussing software artifacts. On the other hand, the developers are forced to interact with computing systems to create and modify software artifacts using low-level text editors and representations designed for compiler input. This forces them to continually shift their focus between levels, causing errors in transcribing intended actions into text and causing conceptually simple actions to be slow and tedious.

The goal of our research is to bridge that gap – to raise the linguistic level of developer/computer interaction. Our hypothesis is that expressing operations on program source code at a level above text-based editing will improve programmer's efficiency and result in fewer errors.

Programmers can benefit most from being able to express operations that entail pervasive large-scale modifications to an existing body of source code. Examples abound in the many maintenance tasks faced by developers during the lifetime of a typical software project. Modification is complicated because traditional text-based programming notations are not easily amenable to change. For instance, simply adding an argument to a procedure requires visiting every invocation site and supplying the missing value. Another example, observed by the aspect-oriented programming researchers, involves delocalized design abstractions such as exception handling, logging, synchronization, and others. Capturing these design abstractions at the source code level is difficult due to limited data and procedural abstractions provided by most programming languages. As trivial a modification as changing which exceptions are handled following the call to a library routine requires finding all such calls and modifying the exception handlers. Yet another flavor of high-level operations is the generative operations that produce chunks of boilerplate code. Outputting fields of a data structure is an example of such an operation. If the list of fields is large and may change over the lifetime of the program, maintaining the output routine "manually" is a tedious and boring task.

Various proposals have been made for systematic modification to existing source code. However, few tools have found their way to the "programming trenches". Our research attacks several major issues with prior approaches: generality, acceptance by the user community, improved abstraction management, and avoidance of proprietary programming language extensions (such as those used in aspect-oriented programming).

Interactive Transformation Environment

When describing their changes to one another, programmers evoke notions such as variables, expressions, statements, loops, and assignments. They also directly refer to names found in source code. These concepts represent the common terminology understood by programmers, making the use of those terms natural for describing actions at a high level. We are prototyping an interactive environment that enables the developer to express source code manipulations using high-level "update scripts" that can be processed by the environment. The scripts might be executed interactively; they might be stored in a library or catalog, or they might serve as update agents bound to program components. A sample update script might look like:

```
for each statement like foo(@args@) replace with foo(<args>, true)
```

This script describes an update following the addition of a new argument to function *foo*.

Another update script of a more generative flavor might be something like:

```
generate method Node.mem_size() as

int mem_size() {
      int result = 0;
      [for each field f of  class Node emit "result += <f>.mem_size();"]
      return result;
}
```

This script acts as an update agent that describes how to generate a method. The generated method will be updated each time a field is added or removed in the class Node.

The examples use a scripting language that embeds both instances and abstractions from the user's source code. The scripting language used here is merely a sample; the actual language to be used must be carefully designed and is part of our research. We draw a clear distinction between the *end-developer* who is well-versed in the application domain and the *tool-builder* familiar with the internals of the programming environment. On the one hand, the scripting language must be expressive; that is, it must provide access to the syntactic and semantic structure of source code. On the other hand, the underlying program representation must not be exposed to end-developers, as it will hamper their ability to formulate transformations. An important factor that distinguishes this research from prior work is that our notation will accommodate both the linguistic abstractions understood by the end-developers and the structural abstractions used by the tool.

The literature on the psychology of programming includes various hypotheses on how programmers construct mental models of programs [Detienne 01]. However, the existing studies are largely concerned with acquiring higher-level schematic knowledge and constructing domain models. Using Java for our initial testbed, we are currently conducting a study of how expert programmers formulate update plans and what forms these plans take. These studies will both contribute to the existing body of research on psychology of programming and allow us to develop a methodology for working with languages other than Java.

Along with the scripting language we are designing an interactive environment for manipulating and executing update scripts. This environment will enable the programmer to visualize execution of the script, examine each transformation site, selectively undo or modify individual transformations, etc. A key aspect of this environment will be the ability to capture the change history of a source code in terms of high-level manipulations. Not only does such a capability

help to document important aspects of program evolution, but also it supports selective rollback of high-level changes days, months, and even years after they had been performed.

Unless our studies suggest otherwise, the environment will augment the scripting language with direct manipulation. We believe that the scripting language will provide the right high-level vernacular for describing code, and we expect professional developers to have little trouble specifying the control structure of pattern matching and transformations in a textual notation. At the same time, we envision a "by-example" pattern matching mechanism whereby the user selects language constructs that "look like" the intended target of the match. The pattern can be subsequently abstracted to match a larger class of source code fragments.

An important advantage of using an integrated environment for transforming source code is the ability to treat the update scripts as abstractions. Not only does this permit naming such scripts and storing them in a transformation library for reuse, it also allows treating scripts as *update agents*. An update agent is a metaprogram bound to both the source and the target (generated) program elements. An integrated environment can track dependencies between the two sections of source code and act appropriately if the developer chooses to make changes to either.

Comparison with Related Approaches

Using automated assistance for manipulating source code is not a new idea. There are numerous systems for source-to-source transformations, program refactoring, and manipulating source code through metaprogramming. However, a number of key issues differentiate prior art from the kind of support for interactive program manipulation that we envision.

A broad class of tools allows the developer to specify transformations in a general-purpose notation that is subsequently processed by the transformation tool. The simplest are primitive text processing tools such as the Unix SED, which offers pattern substitution facilities based on regular expressions. More complicated tools such as LSME [Murphy 95] and TLex [Kearns 91], operate on the lexical structure of the program. However text-oriented tools are not sufficiently powerful and are unable to perform operations based on syntactic or semantic structure.

Many existing full-featured program transformation tools operate on program source code represented as annotated syntax trees. Notable examples include ASTLog [Crew 97], TAWK [Griswold 96], A* [Ladd 95], REFINE [Burson 90], TXL [Cordy 88] and the Intentional Programming environment [Simonyi 95]. Although powerful, tree-based notations exhibit significant usability issues even for expert programmers. Because the structure represented by a tree-based notation does not reflect the user's model of the program structure, such a notation creates a significant barrier for anyone not familiar with the tool's internal representation.

In addition to the already cited problems, one criticism that applies to all of the above program manipulation tools is that they do not facilitate high-level interactive operations. None of the tools include facilities for evaluating or visualizing the pattern matching process, making it difficult for developers to verify that the transformation specification they constructed is complete and correct. These tools offer no way to selectively undo transformations and provide no help for those unfamiliar with the pattern language. There are no abstraction facilities that permit capturing transformations and treating them as update agents or as library components.

Another class of program transformation tools consists of those intended primarily for metaprogramming. Such systems perform compile-time transformations according to a

developer-supplied specification. Many metaprogramming tools operate by performing generic source-to-source transformations. Syntactic and semantic macro processors such as MS [Weise 93], XL [Maddox 89], OpenJava [Tatsubori 00], OpenC++ [Chiba 95] as well as aspect-oriented programming systems such as AspectJ [Kiczales 97] fall into this category. These tools are designed for compiler-like use and have many of the same limitations as other batch-oriented development tools. The result of the transformation is not exposed to the user and hence cannot be manipulated. Most importantly, these tools often work by extending the underlying programming language, which can limit their applicability and also be a deterrent to acceptance by the software industry. A conscious goal of our research is to avoid such extensions, and to demonstrate that the need for language extensions introduced in prior approaches may be lessened using our manipulation abstractions. Our research strives to demonstrate that augmenting conventional language-based systems, rather than inventing new programming language notations can gain much benefit at lesser acceptance cost.

Recently some interactive programming tools have begun to offer restructuring transformation facilities that help reorganize source code in a well-defined meaning-preserving manner. Fowler [99] catalogs a number of restructuring transformations in object-oriented programs (called refactorings). A few of the refactoring transformations have been automated in some widely-used programming environments such as the Refactoring Browser for SmallTalk [Roberts 97], a testament to the usefulness of organizing one's code manipulation behavior in their terms.

However, it is difficult to rely on primitive refactorings for specifying a complex transformation, since transformations frequently depend on the context. For instance, moving a method from *Class1* to *Class2* cannot be fully automated because it is not feasible to automatically locate an appropriate instance of *Class2* in every call context. Yet, a code-specific formulation of the same refactoring, such as moving a method when *Class1* contains a single reference to *Class2* in a public field, is almost trivial. Although cataloging and automating some restructuring transformations is beneficial, attempting to derive an exhaustive list of useful transformations is futile. Much like design patterns in object-oriented programming, refactorings provide a common language for discussing high-level program modifications, rather than prescribing an easily automatable recipe for carrying out the transformations.

Implementation and Evaluation

Our interactive transformation environment is being prototyped using the Eclipse platform, which is an open-source framework for building interactive development tools [Eclipse 02]. The framework provides the infrastructure for building such tools, including editors, source code browsers, project managers, and other IDE-like facilities. Implementing general source-to-source transformation facilities requires a fair amount of program analysis infrastructure. Such infrastructure is not available in the Eclipse platform and will be contributed to Eclipse through integration with the Harmonia program analysis framework [Boshernitsan 01].

Our prototype will be evaluated against several criteria, including (a) current mechanisms for manipulating source code in an interactive setting, (b) usability studies of the resulting prototype as well as evaluation in terms of cognitive dimensions derived from research in cognitive psychology [Green 96], and (c) the outcome of deploying our Eclipse-based implementation in the Eclipse community. The system will be instrumented to collect information about how it is used. It will be possible to selectively disable capabilities. The data we get will not only allow us to learn when aspects of the system are working the way we intend, but to do comparative

studies. We will compare users carrying out a fixed set of modification tasks with and without our tools, and will compare such factors as modification time, user-perceived ease or difficulty of the task, and quality of the resulting transformations.

References

[Boshernitsan 01] Boshernitsan, M. HARMONIA: *A Flexible Framework for Constructing Interactive Language-Based Programming Tools*. M.S. Report. EECS, Computer Science Division, University of California, Berkeley. UCB/CSD-01-1149, 2001. http://www.cs.berkeley.edu/Research/Projects/harmonia/papers/maratb-master.ps.gz

[Burson 90] Burson, S. & Kotik, G. & Markosian, L. A Program Transformation Approach to Automating Software Re-Engineering. In *Proceedings of the International Computer Software & Application Conference 1190*, Chicago, IL, 1990,

[Chiba 95] Chiba, S. A metaobject protocol for C++. In *OOPSLA'95*, 30, 1995, pages 285—299.

[Cordy 88] Cordy, J.R. & Halpern, C.D. & Promislow, E. TXL: A Rapid Prototyping System for Programming Language Dialects. *Proceedings of the International Conference of Computer Languages*, 1988, p. 280-285.

[Crew 97] Crew, R.F. ASTLOG: A language for examining abstract syntax trees. *Proceedings of the First Conference on Domain Specific Languages*, 1997, p. 229—242.

[Detienne 01] Detienne, F. *Software Design: Cognitive Aspects*, 2001, Springer Verlag.

[Eclipse 02] Eclipse Consortium. *Eclipse.org Main Page*. http://eclipse.org/

[Fowler 99] Fowler, Martin (with contributions by Kent Beck and John Brant). *Refactoring: Improving the Design of Existing Code*, Addison Wesley Longman, Inc, 1999.

[Green 96] Green, T. R. G. and Petre, M. Usability Analysis of Visual Programming Environments: A Cognitive Dimensions Framework. *Journal of Visual Languages and Computing 7*, 2, 1996, p. 131-174.

[Griswold 96] Griswold, W. G., Atkinson, D. C., and McCurdy, C. Fast, Flexible Syntactic Pattern Matching and Processing'. *Proceedings of the IEEE 1996 Workshop on Program Comprehension*, 1996.

[Kearns 91] Kearns, S. TLex. *Software Practice and Experience 21*, 8, 1991, p. 805—821.

[Kiczales 97] Kiczales, Gregor, et. al., "Aspect-Oriented Programming". In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag LNCS 1241. 1997.

[Ladd 95] Ladd, D.A. and Ramming, J.C. A*: A Language for Implementing Language Processors. IEEE Transactions on Software Engineering 21, 11, 1995

[Maddox 89] Maddox, W. *Semantically-Sensitive Macroprocessing*. Master's thesis, University of California, Berkeley. UCB/CSD 89/545, 1989.

[Murphy 95] Murphy, G.C. & Notkin D. Lightweight source model extraction. *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering,* 1995.

[Roberts 97] D. Roberts, J. Brant, R. Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems* 3, 4, 1997.

[Simonyi 95] Simonyi, C. *The Death of Computer Languages, the Birth of Intentional Programming*, NATO Science Committee Conference, 1995.

[Tatsubori 00] Tatsubori, M. & Chiba, S. & Killijian, M.-O. & Itano, K. OpenJava: A Class-based Macro System for Java. *Reflection and Software Engineering*, vol. 1826, Lecture Notes in Computer Science, W. Cazzola, R. Stroud, and F. Tisato, Eds.: Springer-Verlag, 2000, p. 117-133.

[Weise 93] Weise, D. & Crew, R. F. Programmable syntax macros. In *Programming Language Design and Implementation (PLDI)*, 1993, p. 156-165.