A Refactoring Tool for Ada 95

Paul Anderson
GrammaTech, Inc, 317 N Aurora St., Ithaca, NY 14850. Tel: +1 607 273-7340
paul@grammatech.com

ABSTRACT
Refactoring is a technique for restructuring code to improve its design. A tool for automatically applying refactoring transformations to Ada 95 programs is described. The tool is based on a language-neutral static-analysis toolset named CodeSurfer, and uses the ASIS interface to retrieve basic facts about a program. A higher-level object-oriented interface to the ASIS representation is described, which enables pattern-matching queries on the program's abstract semantic graph. The tool has been validated by using it to apply a set of transformations designed to bring a general-purpose code library into compliance with a safety-critical subset of Ada. Future plans for a set of tools based on these techniques are described.

Categories and Subject Descriptors
D.2.6 [**Software Engineering**]: Programmer workbench, Ada.

General Terms
Languages.

Keywords
Refactoring, Ada 95, ASIS, Static Analysis, Software Engineering Tools.

1. INTRODUCTION
Refactoring is a technique for restructuring software by applying a series of transformations that make the code easier to understand and maintain, but which do not change its essential behavior. This paper describes an experimental tool for automatically applying refactoring transformations to Ada code. The remainder of this paper is structured as follows. Section 2 gives some background material on refactoring and on CodeSurfer—the infrastructure on which the tool was built [2, 3]. Section 3 describes how ASIS was used as the basis for the tool. Section 4 discusses the design of the code transformation part of the tool. Section 5 describes how the tool was validated by using it to automatically perform a number of refactoring transformations on the Charles library to bring it into compliance with the SPARK subset. Section 6 describes related work. Finally, Section 7 summarizes and describes our plans for future development of the tool.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGAda'04, November 14-18, 2004, Atlanta, Georgia, USA.
Copyright 2004 ACM 1-58113-906-3/04/0011…$5.00.

2. BACKGROUND
2.1 Refactoring
Refactoring is a technique for restructuring software by applying a series of transformations that make the code easier to understand and maintain, but which do not change its essential behavior. Cunningham and Beck identified refactoring as a useful technique for improving programmer productivity and code reuse in the 1980s while working with Smalltalk [7]. Refactoring has since become an important technique in the Smalltalk community. The most significant recent work is described in Fowler's book [9], which catalogs a set of refactoring transformations. Refactoring has become an important component of the *Extreme Programming* methodology [6], but is sufficiently general that it is useful in many other methodologies.

A typical entry in the catalog of refactoring transformations describes the motivation for applying the transformation, and the mechanics of applying it. For example, one transformation is the *Extract Method* transformation. The motivation is to simplify the code by factoring out fragments of code that can be grouped together into a new method or procedure. The following Ada code fragment is a candidate for this transformation:

```
procedure Print_Pay_Check
     (Fed_Rate   : in Float;
      Local_Rate : in Float) is
begin  -- Print_Pay_Check
   Put("Pay to ");
   Put(Name);
   Newline;
   Put("Amount: ");
   Put(Gross - Gross * Fed_Rate -
       Gross * Local_Rate);
   Newline;
end Print_Pay_Check;
```

In this example, the computation of the amount of payment is mixed in with the printing of the check. A better design would be one that computed the amount in a new method. Hence the Extract Method refactoring transformation could be applied to produce the following code:

```
function Compute_Amount
      (Fed_Rate   : in Float;
       Local_Rate : in Float)
      return Float is
begin  -- Compute_Amount

   return Gross - Gross * Fed_Rate -
          Gross * Local_Rate;

end Compute_Amount;
```

```
procedure Print_Pay_Check
      (Fed_Rate   : in     Float;
       Local_Rate : in     Float) is
begin  -- Print_Pay_Check
   Put("Pay to ");
   Put(Name);
   Newline;
   Put("Amount: ");
   Put(Compute_Amount(Fed_Rate,
                      Local_Rate));
   Newline;
end Print_Pay_Check;
```

Of course, the transformation is more than simply copying the text of the code from one place to another. There are a number of requirements that must be met before the code can be moved. For example, the user must be careful to pass the right parameters to the new method, and to make sure that local scope variables in the factored-out code are handled correctly. In the above example, note that the variables `Fed_Rate` and `Local_Rate` are in a scope local to the original procedure, so they must be passed as parameters, whereas the variables `Name` and `Gross` are in a scope enclosing the procedure, so these can be used as-is because they will still be in scope in the new procedure.

2.2 CodeSurfer

The tool was developed using the CodeSurfer infrastructure for program manipulation and analysis [2, 3]. This section briefly describes this tool. CodeSurfer is a whole-program static analysis tool. Given a program, CodeSurfer will produce a number of intermediate representations of that program. These include the program's abstract syntax tree (AST), abstract semantic graph (ASG), control-flow graph (CFG), call graph, points-to graph, and system dependence graph (SDG). Production of each of these intermediate forms is optional. For the refactoring application described herein, the only form required was the ASG, although planned improvements will also require the CFG.

CodeSurfer provides a range of analysis services on each of these representations. Pattern matching on fragments of the AST and ASG is provided. Services on the CFG include model-checking techniques for reasoning about the possible paths through the program. Slicing and chopping queries are available on the program's SDG.

A scripting language based on Scheme is provided to allow users to author specialized analyses and transformations. Scheme is a simple interpreted functional language similar to Lisp.

An advanced graphical user interface (GUI) allows a user to browse these intermediate forms and pose queries. CodeSurfer is thus named because it allows a user to surf their programs in a manner akin to surfing the web.

CodeSurfer has been available for some time for C and C++. This project represents the first fruit of an effort to target it to Ada.

3. ASIS INTERFACE

Some of the more simple refactoring transformations can be performed using text-editing tools, and some others can be implemented as transformations on the program's abstract syntax tree. However, for full generality, the refactoring tool must have access to the abstract semantic graph for the entire program. Ada tool writers are fortunate to have the ASIS interface for enabling these kinds of applications [4].

From the point of view of the refactoring tool, the ASIS interface is very low level. Instead, we preferred to use the much more expressive interface to the ASG provided by CodeSurfer. This interface has the following advantages over the raw Ada ASIS interface:

- It is object oriented, and allows metaclass operations. ASTs lend themselves naturally to being expressed in an object-oriented manner. This enables many operations to be expressed succinctly.
- It has sophisticated pattern-matching primitives. This allows easy specification of functionality to search for opportunities for refactoring.
- It is accessible through the Scheme scripting language. This allows the system to be extended quickly and easily by the end user.

Furthermore, the CodeSurfer interface allows for access to the higher-level intermediate representations such as the control-flow graph and the dependence graph. Although these are not necessary for the refactoring transformations described here, they will be useful for performing advanced queries such as those described in Section 7.

The CodeSurfer ASG interface consists of three layers on top of the Ada ASIS interface. These are described in the following sections.

3.1 C interface to ASIS

The first layer simply provides an interface to the ASIS library in C. This was required for the sole reason that CodeSurfer Scheme implementation is written in C. In this interface, the core ASIS types are represented as opaque C structures. The enumerated types are represented by constants in C. The ASIS `List` types are represented as vectors in C. Each ASIS subprogram is represented as a similarly named subprogram in C. The Ada `Wide_String` type is represented as `wchar_t*` in C. Subprograms with a single default parameter in ASIS are represented as two separate C functions, one with the parameter present, and one without. Subprograms with more than one default parameter are represented by a set of functions, as a simple generalization of the single default parameter case.

3.2 Scheme interface to ASIS

The second layer is a Scheme interface to ASIS. In this interface, the core ASIS types are represented by primitive Scheme types. The ASIS List types are represented by Scheme lists. Enumerated types are represented by Scheme symbols. Each ASIS subprogram is represented by a Scheme function. This interface is very similar to the ASIS interface. Many ASIS programs can be easily transcribed to their equivalent in Scheme using this interface.

3.3 Class-based interface to ASIS

The final layer provides a class-based interface to ASIS. Abstract syntax trees lend themselves naturally to being represented in an object-oriented fashion, and the ASIS specification is no different. Consider the representation of a simple constant expression `1.0` in ASIS. In ASIS all program constructs are represented using the type `ASIS.Element`. Objects of this type are categorized first

by the enumeration type Element_Kinds. Our simple expression has kind An_Expression, and can be represented as type ASIS.Expression, which is a subtype of ASIS.Element. The enumeration type Expression_Kinds further categorizes expressions. Our expression has kind A_Real_Literal.

Although expressed in a non-object oriented way in ASIS, this representation maps over to a class-based representation very easily. As our Scheme implementation provided a CLOS-style class interface, we are able to represent the entire ASIS interface in an elegant object-oriented way.

In this interface, there is a class for each of the two main ASIS types: ASIS.Element and ASIS.Compilation_Unit, respectively named asis:an-element and asis:a-compilation-unit. For each subkind of these, there is a subclass. For the example above, the constant expression is represented as an instance of class asis:a-real-literal, which is a subclass of asis:an-expression, which is a subclass of asis:an-element.

Each class in this hierarchy has a (possibly empty) list of attributes and children. The children are simply the children in the AST. An attribute of a class is any other value associated with the node. Most ASIS queries on elements are represented in this class hierarchy as children or attributes.

For example, the ASIS interface defines the function ASIS.Elements.Enclosing_Element. This function returns an element's parent in the AST. This is represented as an attribute named enclosing-element on the class asis:an-element. In Ada, one would retrieve this by the call:

```
ASIS.Elements.Enclosing_Element(Item);
```

In our class-based interface in Scheme, the element can be retrieved by the call:

```
(ast-attribute item :enclosing-element)
```

This class-based interface is much more expressive than the simple Scheme interface or the Ada interface itself. Neither of the latter allow pattern matching on the ASIS trees, and neither have the meta-class interface. For example, in the Ada interface, it is very difficult to write a program that retrieves all of the attributes associated with an arbitrary element. Such a program requires the programmer to understand every aspect of the ASIS interface in advance. The code would have to be structured as an enormous nested switch statement, which would make it difficult to write and harder to understand.

In contrast, such a program in the class-based interface to ASIS is trivial. The following function creates a list of all the attributes of the given element and returns it.

```
(define (get-attrs el)
     (map (lambda (kw)
              (ast-attribute el kw))
            (ast-attributes el)))
```

A brief explanation of this is worthwhile. The function being defined is named get-attrs, and it takes a single parameter el, which is expected to be an element. This function calls the map function with two parameters: the closure given by the lambda expression, and the list returned by the call (ast-attributes el), which is the list of attribute names associated with the element. The function map applies the closure to each of these attribute names in turn (the parameter kw) and returns a list consisting of the value of the attribute given by (ast-attribute el kw).

The ability to express this kind of functionality succinctly is extremely useful. For example, we use methods much like this to implement a program to allow a user to visualize and navigate the ASIS representation of a program.

The other part of the interface that makes it highly expressive is its pattern-matching capabilities. Consider a subprogram that looks for occurrences of statements of the form X := X + 1; for any scalar variable X. This kind of subprogram is awkward to write in ASIS. The author of the code must make an instance of Asis.Iterator.Traverse_Element using a procedure to visit each element. That procedure must first test whether the element being visited has subkind An_Assignment_Statement. It must then retrieve the children of that node, and determine if the second child is of subkind A_Function_Call. The children of that element must then be retrieved and their kinds checked. To determine if the + operator is used, the function Prefix must be called and the result must have subkind A_Plus_Operator.

This tedious retrieval of the children and their comparison of the kinds must continue until the leaves of the tree have been inspected. Finally, the code must compare the name on the left-hand side of the assignment statement with the name given as the first parameter to the call on +.

In contrast, the following Scheme code specifies a pattern that can be used in a traversal to find occurrences of the same statement:

```
((asis:an-assignment-statement
   x
   (asis:a-function-call
    (asis:an-operator "+")
   ((asis:an-identifier x)
    (asis:an-integer-literal 1)))
```

The x in the above pattern is a pattern variable. The first occurrence of a pattern variable is its *binding* occurrence. Subsequent occurrences are *use* occurrences. The pattern will match only if all use occurrences are equivalent to the binding occurrence.

This comparison is not to denigrate ASIS. ASIS is a very well designed interface. The point is that the highly dynamic nature of Scheme enables a class-based interface that provides a higher level of abstraction. This in turn enables rapid prototyping of source-code analysis tools like the one described here.

4. CODE TRANSFORMATION

The goal of the tool was to provide a way of transforming code from one form to another. Given that the code is represented as an ASIS tree, one approach to this would be to encode the transformation as a tree-to-tree transformation. The user would search for subtrees where transforms must be applied, create a new subtree to replace the selected one, then render, or prettyprint, the tree as a text file again. This approach has the advantage that tree-to-tree transformations can be type checked, so it is possible to guarantee that no transformation can possibly introduce any syntactic or static-semantic errors. However, this

approach is impractical. One problem with it is that the ASIS trees are read-only. This limitation could be avoided, but another more important limitation of the approach then presents. This is that it is difficult to prettyprint ASIS trees in a manner that minimizes the changes to the original code.

The layout of code, which comprises both comments and whitespace, is extremely important to an end user. Judicious use of whitespace improves readability. Comments in code are a key asset. Neither should be blithely discarded or moved. Any tool that did so would have difficulty being accepted by users.

The approach we favor, and which is used in the tool, is one that transforms code by applying a sequence of textual edits on the file. When a set of subtrees is located where the refactoring should be applied, the transformation is encoded as a set of annotations on the nodes in the subtrees. Each annotation can encode:

- Text to be placed before the associated node
- Text to be placed after the associated node
- Text to replace all of the associated node's original text.

The new textual representation is then produced by traversing the tree and interpreting the annotations at each node with respect to the original file. This addresses the problem with retaining whitespace and comments, because as much of the original file and its layout is retained as possible.

There is an inherent risk to this approach, which is that textual edits increase the possibility of introducing syntax errors into the code. For example, consider a refactoring transformation that removes a parameter from a call to a subprogram. It is not good enough to simply delete the text associated with the parameter, as there may be a comma before or after the parameter that also must be deleted. The author of a refactoring transformation must be careful to consider all the contexts in which an edit may be made and allow for cleaning up the syntax appropriately. In practice, we have found that this is not difficult. In any case, the risk of this introducing a syntactic error that is not immediately caught by the compiler is very low.

Another disadvantage is that if several transformations require that a number of different edits be applied to the same subtree, then the sequencing of those edits is crucial. In this situation, the tool may call for further direction from the user. If this is infeasible, then the user can always apply the transformations one at a time in sequence.

5. VALIDATION

In order to validate our approach, we chose to apply it to a problem faced by many Ada users. There are several safety-critical subsets of Ada, including the SPARK subset, and the similar ZBRA subset used at Boeing. Projects that wish to reuse code from another project in a safety-critical project that uses one of these subsets often find that the code does not conform to the rules of the subset. Finding places in the code that are outside the subset, and modifying them appropriately is a major task. Some of the changes require much thought to make them compatible with the subset. However, many others are quite simple and repetitive. Having a tool to perform such transformations automatically would relieve the engineers of a great deal of drudgery, give them confidence that the transformations were performed correctly, and would free them to concentrate on more appropriate tasks.

In collaboration with Boeing, we identified several such transformations and were successful in implementing a refactoring environment capable of applying them to existing code. We were able to apply the tool successfully to a large body of code.

The rules we identified are from the SPARK subset:

1. Default parameters to subprograms are prohibited. For example, if there exists a function with the signature:

   ```
   function F(X : in Integer;
              Y : in Integer := 0) ...
   ```

   which has calls of the form:

   ```
   Result := F(100);
   ```

   then the declaration should be transformed into:

   ```
   function F(X : in Integer;
              Y : in Integer) ...
   ```

   and the call should be:

   ```
   Result := F(100, 0);
   ```

2. All parameters must be passed using named associations. The above call, which uses positional parameter associations, should be transformed into:

   ```
   Result := F(X => 100, Y => 0);
   ```

3. All use clauses are prohibited. For example, if a package has

   ```
   use Text_IO;
   ```

   and if a statement in that procedure contains a call to a function in package Text_IO:

   ```
   Put(Output);
   ```

   then the use clause should be removed, and the statement should be transformed to:

   ```
   Text_IO.Put(Output);
   ```

A screenshot of the tool in action is shown in Figure 1. The graphical user interface shown offers the following features:

1. The user can select which refactoring transformations to apply, and on which compilation units.
2. The transformation may apply to many files. The changes to each file can be reviewed in turn.
3. Each proposed change can be omitted simply by clicking on the text that is to be changed.
4. Each file can be written out manually. However, there is also a "Batch" command that makes all the changes to all files and writes them out to disk.

The tool also has a non-graphical user interface that allows the user to apply it to large bodies of code without any further input or intervention from the user.

5.1 Application to Charles

In order to demonstrate the viability of our approach, we chose to use it on some real-world code.

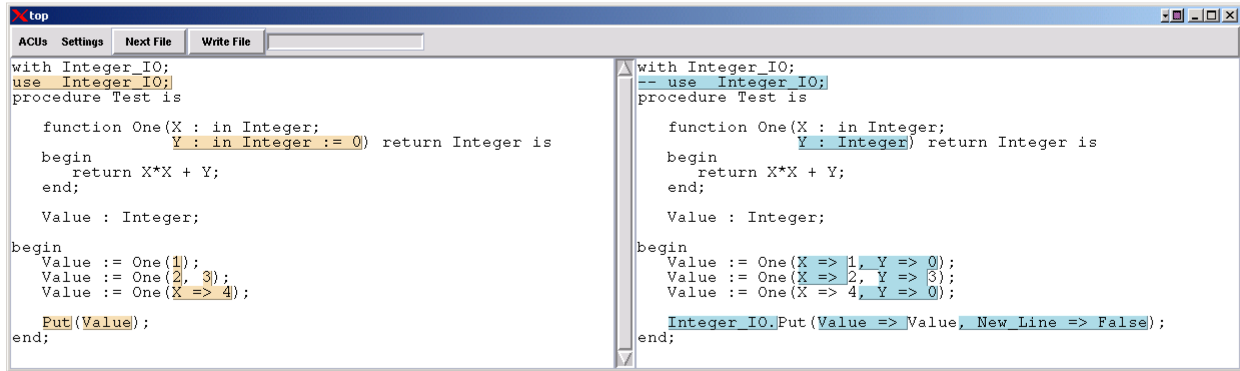

Figure 1. *The prototype tool in action. The pane on the left shows the original program. Parts of the program that must be changed are highlighted. The pane on the right shows the program after the changes have been applied. Any single transformation can be undone by clicking on it in the right pane.*

The *Charles* library is an open-source library of general-purpose data structures, comprising approximately 45,000 lines of code over 167 files [10]. We chose to apply the tool to this code not because we advocate using Charles in a safety-critical application, but simply because it represented a nicely-sized body of reusable code.

The refactoring tool was able to perform all the refactoring transformations on this code in less than two minutes on a modestly configured PC running Linux with multiple users. As none of the techniques and algorithms used in the system are super-linear, this shows that large Ada programs are well within reach of the system.

This exercise turned up one interesting situation where application of one of the SPARK rules resulted in a non-compilable program. The rule in question is the one that specifies that named associations must always be used in calls. The following example distils the essence of the issue. Charles defines a procedure that is defined in two ways:

```
procedure Proc(L : in out T1;
               R : in out T2);
procedure Proc(R : in out T2;
               L : in out T1);
```

Both procedures have exactly the same essential semantics—the two variants are provided simply as a convenience to the user. Given two variables `V` and `W` of types `T1` and `T2` respectively, then the call `Proc(V, W);` which uses positional association will resolve to the first of these.

However, no call to either of these can be written using named associations for the parameters: the call

```
Proc(L => V, R => W);
```

matches both procedures equally well, so the compiler cannot resolve the call and reports the error.

6. RELATED WORK

Cunningham and Beck identified refactoring as a useful technique for improving programmer productivity and code reuse in the 1980s while working with Smalltalk [7]. Refactoring has since become an important technique in the Smalltalk community. Opdyke's thesis in 1992 explored applying refactoring to a C++ development environment, presented a set of semantics-preserving transformations for refactoring, and discussed tool support for refactoring [11]. The most significant recent work is described in Fowler's book [9], which catalogs a set of refactoring transformations. Refactoring has become an important component of the *Extreme Programming* methodology [6], although it stands on its own as a highly useful technique.

There are a small number of interactive refactoring tools for a variety of languages. Refactoring was initially developed for Smalltalk. The pre-eminent tool for Smalltalk is *Refactoring Browser* for IBM VisualWorks [8]. For Java, the most advanced refactoring tool is part of the Eclipse toolset [1].

Semantic Designs Inc. offer DMS, a toolkit for software reengineering capable of performing non-interactive transformations on large bodies of code [5].

No tools are known that provide any support for interactive refactoring of Ada programs.

7. CONCLUSIONS AND FUTURE WORK

We have developed a new tool for performing refactoring transformations on Ada 95 and have demonstrated its viability by applying it to existing code to help adapt existing reusable code to new purposes.

The work described here is the fruit of a six-month feasibility study. A new two-year effort has been started to extend the capabilities of the tool. A suite of three related tools is planned. A *refactoring assistant* will scan code looking for opportunities to apply transformations based on user-specified goals. A *refactoring editor* will apply selected transformations to the code. A *refactoring manager* will keep track of the state of the program and allow a user to back out of any transformations that prove problematic.

The tool will provide a wide range of refactoring transformations, including a selection of those from the Fowler catalog. It will also be extensible and programmable so that users can author their own refactoring transformations. We will actively seek input from our industrial partners and the Ada community to ensure that the tool addresses the needs of Ada developers.

This project is part of a broader effort to adapt the CodeSurfer suite of tools to Ada 95. When complete, CodeSurfer-based tools currently available for C/C++ will be available for Ada 95. This includes source-code understanding tools, and tools that exploit

model-checking techniques for automatically finding programming flaws.

8. ACKNOWLEDGMENTS

The author wishes to thank Haakon Larsen for his work on this project. This work was partially funded by MDA contract DASG60-03-P-0054.

9. REFERENCES

1. The Eclipse Project, http://www.eclipse.org/eclipse/index.html.
2. CodeSurfer User Guide and Reference Manual, http://www.grammatech.com.
3. Anderson, P., T. Reps, and T. Teitelbaum, *Design and Implementation of a Fine-Grained Software Inspection Tool.* IEEE Transactions on Software Engineering, 2003. **29**(8): pp. 721-733.
4. Anonymous, *ASIS 2.0.D specification.* http://www.acm.org/sigada/WG/asiswg/asiswg.html.
5. Baxter, I., The DMS® Software Reengineering Toolkit, http://www.semdesigns.com/Products/DMS/DMSToolkit.html.
6. Beck, K., *Extreme Programming Explained: Embrace Change.* 1999, Reading, MA: Addison Wesley.
7. Beck, K. and W. Cunningham, *A Laboratory for Teaching Object-Oriented Thinking.* In *OOPSLA '89.* 1989. New Orleans, LA. pp. 1-6.
8. Brant, J., *Refactoring Browser.* 2001: http://st-www.cs.uiuc.edu/users/brant/Refactory/.
9. Fowler, M., *Refactoring. Improving the Design of Existing Code*. 1999, Reading, MA: Addison Wesley.
10. Heaney, M., The Charles Container Library, http://charles.tigris.org.
11. Opdyke, W.F., *Refactoring Object-Oriented Frameworks.* Ph.D. Dissertation *Computer Science Department*. 1992, University of Illinois at Urbana-Champaign.