A Program Transformation Approach to Automating Software Re-engineering

Scott Burson, Gordon B. Kotik and Lawrence Z. Markosian
Reasoning Systems, Inc.
3260 Hillview Avenue
Palo Alto, CA 94304

Abstract

We describe a new approach to software re-engineering that combines several technologies: object-oriented databases integrated with parsers for capturing the software to be re-engineered; specification and pattern languages for querying and analyzing a database of software; and transformation rules for automatically generating re-engineered code. We present a program transformation system, REFINETM, that incorporates these and other technologies within an integrated environment for software re-engineering. Finally, we present examples of how our approach has been applied to re-engineering software in a variety of languages.

1 Overview

Eighty percent of programming resources are allocated to maintaining and re-engineering existing code, not to developing new applications [3]. The need for specialized re-engineering and maintenance tools is driven by the need to analyze and understand a large body of existing code and then modify it so that it conforms to new requirements. The existing code may be written in old languages or dialects, may run on obsolete hardware, or may use an old database management system instead of a modern relational one that may be more appropriate.

Development of new software requires traditional CASE tools such as incremental compilers, interpreters, diagrammers, simulators and code generators. Re-engineering, on the other hand, requires tools that capture, analyze and transform existing code. This paper describes a CASE approach to automating maintenance and re-engineering that applies, in a novel way, new and traditional computer science technologies to these problems. Our approach uses

- object-oriented databases integrated with LALR parsing technology to capture and represent software as annotated abstract syntax trees stored in a database;
- a program specification and pattern matching capability to describe and analyze software once it is in the database; and
- program transformation rules to automatically perform large-scale modifications to the software.

Typical analysis and transformation tasks might include the following:

- Analysis: What parts of my C program might behave differently when ported to a new platform? What datasets are input or output by which COBOL programs in my JCL code?
- Transformation: Make the necessary changes to parts of my C program so it will compile on a small computer. Change a program so that I can use the new X window system instead of OldWindows.

Programmers currently store software in text files and apply text-oriented tools to these files in order to address analysis and transformation problems. Examples of these text-oriented tools are editors and string search and replace utilities [2]. The string search utilities generally pattern-match text strings against regular expressions that define the target pattern.

While the string-based facilities provide an improvement over visual search and manual replacement, they do not directly support analysis based on the syntax and semantics of the programming language. In addition, they do not support automatic transformation of complete programs.

This paper presents an approach to machine capture, analysis and transformation that significantly extends
the opportunity for automating maintenance and re-engineering. This approach is centered on:

1. an object-oriented database instead of text files to capture existing software;
2. data definition and query languages that support program analysis;
3. syntactic pattern-matching against program templates expressed largely in the target language (instead of matching text strings against regular expressions);
4. program transformation rules that automatically modify existing code or generate new code.

To integrate this approach with existing file-based storage of software, tools are needed for parsing source files into the database and printing the database back to source files. To facilitate program analysis, the data definition and query languages should support the mathematical abstractions used in high-level descriptions of programs (set notation, first order logic, tree comparison). Pattern matching and transformation are actually performed on the (annotated) abstract syntax of the software, which reflects program structure. Thus the purpose of the object-oriented database is to represent the abstract syntax of the programs, together with any additional information needed for analysis. Sophisticated analysis and code generation tools such as compilers must create their own abstract syntax trees (ASTs) from the text file representation of programs. Our approach facilitates development of comparably sophisticated program analysis and manipulation tools for re-engineering by using an object-oriented database to represent ASTs. The approach is implemented in REFINE, a toolset for software re-engineering.

Section 2 examines the REFINE-based approach to re-engineering. Section 3 considers actual applications of the approach. Section 4 provides a summary and conclusion.

2 The REFINE toolset for software re-engineering

The objective of the REFINE [1] toolset is to provide a foundation for rapid development of automated, language-specific and even application-specific re-engineering capabilities. Language-specific tools are needed because there are many tasks common to representing and analyzing software in a particular language. Such tasks include parsing, control and data flow analysis, pattern-matching and printing. Therefore language-specific tools are likely to be used regardless of the goals or details of the re-engineering task.

Language-specific tools alone cannot achieve a high level of automation in re-engineering, as large-scale re-engineering tasks require application-specific tools. For example, converting a C application containing one vendor's embedded SQL to another SQL may require analysis and modification of both the C and embedded SQL code to implement optimizations appropriate to the target dialect. The extent of the required optimizations will depend on the run-time characteristics of the SQL implementations and the C application, which may not be adequately understood until well into the project.

REFINE provides basic capabilities for representing software, together with generic tools and tool generators. The principal components of the REFINE toolset are:

- a very high level, executable specification language for describing and transforming software;
- an object-oriented database that provides the necessary abstractions for representing software and software-related objects; and
- a language processing system that accepts definitions of programming languages and produces language-specific tools.

Additionally, REFINE provides an X Windows-based graphics package that supports interactive displays of the contents of its database and the results of code analysis.

Programs are converted between source files and the object-oriented database using the parsers and print- ers created by the language processing system. The database is thereby integrated with conventional file-based systems and tools. The REFINE object system and other high-level data types in the specification language (sets, sequences, maps, etc.), support a data model for software objects that captures the standard conceptual view of annotated abstract syntax trees.
Tools that analyze and transform software in the database are written in the REFINE specification language, which provides mechanisms for template-based program description and rule-based program transformation.

Figure 1 indicates how REFINE components can be used in software maintenance.

2.1 The REFINE specification language

The REFINE language supports a variety of specification techniques including set theory, first order logic, rules, object-oriented and procedural programming. The specification language is also used as the query/update language for the database. The compiler for the specification language is implemented as a rule-based program transformation system. The compiler and most of the rest of REFINE are written in this specification language.

Use of the specification language shortens and simplifies program analysis functions. Examples are provided in Section 3.

2.2 Object-oriented database

The REFINE database provides persistent storage of software and software-related objects. It provides mechanisms for version control, concurrency control for multiple users, computed attributes, and constraint maintenance. The database is integrated with the language processing system (described below) and is used to store software in the form of annotated abstract syntax trees. It is customizable by the user for specific languages and applications. In a typical software re-engineering application, the developer will use an existing domain model or define one for the language of the source code to be re-engineered and for the target language if different from the source. The example in Figure 2 shows the definition of a domain model for a simple numerical expression language.

```
classEXPR
  subclass-of USER-OBJECT
  attributes
    TEST-SUCCEEDED?: boolean

class NUMERAL-EXPR
  subclass-of EXPR
  attributes
    MAGNITUDE: integer

class SUBTRACTION-EXPR
  subclass-of EXPR
  attributes
    MINUEND: EXPR
    SUBTRAHEND: EXPR

class MULTIPLICATION-EXPR
  subclass-of EXPR
  attributes
    FACTORS: seq(EXPR)
```

Figure 2: Domain model for Numerical Expression Language (NEL)

Object classes have been defined to model the non-terminals of the language such as expressions. The slots (called attributes in REFINE) model those properties of the non-terminals that are of interest, for example, for parsing, printing, and analysis. The REFINE database is customizable: the user can define domain models for the languages of interest or extend existing language domain models (for example, by adding attributes that annotate the abstract syntax tree). REFINE provides utilities for traversing abstract syntax trees stored in the database.

2.3 Language processing system

The REFINE language processing system takes as input a description of a language in the form of a grammar. It produces a parser, printer, pattern-matcher and mouse-sensitive text browser for the language. Figure 3 is a grammar for the numerical expression language whose domain model was specified in the previous section.
Grammars are written using a high-level syntax description language that includes:

- regular right-part operators for productions;
- precedence tables;
- semantic actions for productions; and
- a mechanism for specifying lexical analyzers.

These capabilities reduce the need for recursive productions and shorten and increase the readability of the grammar specification. A key component of the language processing system is an LALR(1) parser generator. In addition, REFINE provides a capability for defining program templates. These templates can be used:

- in pattern matching, to test whether an existing program is an instance of a template, and
- in pattern instantiation, to build a new program that is an instance of the template.

Typically pattern matching is done during analysis — for example, to detect code that must be altered; pattern instantiation is then performed to generate the modified code. Usually pattern matching is combined with semantic program analyses that exercise the full REFINE specification capability.

The additional production rules needed for parsing program templates are added automatically to the user-specified grammar by the language processing system. Compiling the NEL grammar shown in Figure 3 allows examples of program templates from the example language NEL in Figure 4 to be parsed into abstract syntax trees, and allows the trees to be printed as text. Figure 4 also shows examples of program templates from the example language NEL that are accepted by the parser. Templates are delimited by single quotes.

![Figure 3: Grammar for NEL](image)

<table>
<thead>
<tr>
<th>NEL expressions</th>
<th>Templates for NEL expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 \times 3 - 4 \times 5)</td>
<td>('Ox - Gy' matches (a, c))</td>
</tr>
<tr>
<td>(2 \times (3 - 4) \times 5 \times 12)</td>
<td>('.. * 5' matches (b))</td>
</tr>
<tr>
<td>(2 \times 3 - (4 \times 5))</td>
<td>('Ox - 4 * 5' matches (c))</td>
</tr>
</tbody>
</table>

![Figure 4: Examples](image)

The language processing system has been used to build software management and re-engineering tools for a number of languages, including REFINE itself, Ada, C, COBOL, IBM JCL, SQL, NATURAL and SDL. These tools have in turn been used for applications including automated software maintenance, re-engineering, code generation and program verification.

### 3 Examples

Several examples have been selected to illustrate the application of the database/transformational approach to software re-engineering. Each of these example applications was developed quickly once the required language was modeled and its syntax defined. The syntax definitions were generally derived from a standard published language grammar. The primary technical reason for the speed of development is that we used the very high level language features of REFINE, which allowed us to express concepts about software declaratively. The REFINE compiler then automatically generated code that implemented analysis tools based on these concepts.

#### 3.1 Analysis Examples

An abstract syntax-based representation of software for analysis is commonly used in modern approaches to software engineering. Most software analysis tools build such a model as a preliminary phase to the actual analysis phase. Perry summarizes some of the important uses of abstract syntax trees analysis in software development in [6]. The analysis capabilities and applications described here can be regarded as extensions of...
these common analyses in that they take advantage of the novel features of the database model of software.

The analysis problem addressed in this example is portability of a C program. The C language does not specify the order of evaluation of arguments in function calls. Thus, if a function call has two or more arguments, one of which is a side-effecting expression, the program behavior may differ with different C compilers. Our objective was to analyze programs for this potential portability problem. Our approach was to use a complete language model and syntax for ANSI C that had already been defined in REFINE, and to write program specifications that would identify suspect function calls.

The rule Analyze-Function-Call applies the test to an object in the database and if the test succeeds, puts the function call in a set of program expressions for later examination by the user or for further automated analysis.

Rule Analyze-Function-Call (node)

node = 'SE OF(args)' &
num-args = size(args) &
num-args > 1 &
se1 in [1..num-args] &
  Pure(args(se1))

->
node in *suspect-nodes*

The function Pure returns True if its argument is a non-side-effecting expression. It is defined recursively with a number of base cases, several of which are shown in the following partial definition:

Pure function(expr: expression) : boolean
computed-using
  Literal(expr) or
  Identifier-reference(expr) or
  (expr = 'SE OF(base)[offset]' &
   Pure(base) & Pure(offset)) or
  (expr = 'SE OF(func($args) &
   Pure(func) &
   for-all(arg)(arg in args =>
   Pure(arg))) or ...

We note several points about these analysis functions. First, they are largely declarative specifications, using set-theoretic data types (sequences and sets) and first order logic, including quantifiers. The functions contain no implementation details. Second, where appropriate, they use program templates for pattern matching and pattern variable binding.

For example, the template ‘SE OF(func($args)’ is used to test whether expr is an expression that is a function call. SE is part of the pattern syntax that asserts that what follows is an expression.

If the pattern match succeeds, the pattern variables func and args are appropriately bound and further tests are made to determine whether they are non-side-effecting.

3.2 Dataflow analysis of IBM JCL

The motivation for this example was the need to maintain applications controlled by IBM Job Control Language (JCL). Both the applications and the JCL code itself were undocumented and the authors unavailable. The initial problem faced by anyone maintaining or modifying such code is to understand dataflow among the programs and datasets referenced by the JCL. This section discusses a REFINE application that provides a facility for parsing and analyzing programs written in JCL under the VM and MVS operating systems on a wide range of IBM mainframes. The complete application includes the following components:

- a domain model that specifies the classes of syntactic objects in JCL source code and the relations among them;
- a grammar for parsing and printing JCL source files;
- a data flow analysis tool; and
- code to display, using the REFINE user interface tools, the abstract syntax and surface syntax of a JCL source file as well as the results of the data flow analysis.

The remainder of this section focuses on the REFINE code for the first three of these components. A screen dump is included to illustrate the use of the fourth.

Domain model: The domain model includes an object class for each non-terminal of the JCL language. The model forms a hierarchy under the subclass relation. At the top of the hierarchy is JCL-OBJECT. Part of the hierarchy is shown in Figure 5.

Each class in the hierarchy has one or more attributes (slots), which are expressed as mappings in REFINE. The domain of an attribute map is the object class for which it is defined, and the range is either an object class or data type. The attributes of the DD-STATEMENT class are inherited from STATEMENT, of which it is a subclass:

```
stmt-name: map(statement, general-identifier)
```
REFINE allows an object class to have attributes that are not used for parsing or printing. Such attributes can be used by analysis functions. For example, following parsing, the JCL code might be analyzed to determine, for each statement, the number of parameters it contains. This value could be held in the attribute parameter-count:

parameter-count: map(statement, integer)

Those attributes of an object class that are used to define the abstract syntax tree are asserted to be tree attributes, as in the following example:

define-tree-attributes(DD-statement, { "stmnt-name", "stmnt-parameters" })

REFINE provides database utility functions that traverse and otherwise manipulate database structures for which a tree structure is defined.

Grammar: Once the abstract syntax for a language has been captured in a domain model, a surface syntax can be defined. REFINE provides several tools for defining the surface syntax. For JCL, a customized lexical analyzer was necessary. The lexical analyzer returns the token "statement" whenever it detects a statement number. A parser for JCL was also necessary. REFINE provides a regular right-part production language for defining LALR(1) grammars. The following production defines the surface syntax for the JCL DD statement:

DD-statement ::= [ "statement” { stmnt-name } "DD"stmnt-parameters*", ]

This production says that a DD-statement is of the following form: the keyword "statement" (returned by the lexical analyzer) optionally followed by the name of the statement followed by the keyword "DD" followed by the statement parameters. If there are more than one statement parameter, they are separated by commas. Since stmnt-parameters is defined in the domain model as a sequence of non-null-parameter objects, a (separate) production rule must be included in the grammar for printing instances of the non-null-parameter class. The "[" and "]" that enclose the right part of the production rule for the DD-statement require that each of the items within the square brackets must appear (except for the optional items), and they must appear in the order indicated. The "{" and "}" notation indicates that the enclosed item(s) are optional and may appear in any order.

REFINE also provides mechanisms for defining operator precedence and non-standard semantics for a language. Use of these mechanisms, plus the "*" and related notation used in other rules, simplifies the grammar definition and avoids the need for most recursive productions.

When the grammar definition is compiled, REFINE generates a parser, printer and pattern matcher for the language. Figure 6 contains an example of JCL code that can be read into the REFINE object-oriented database, together with a diagram of the abstract syntax of the sample.

Data flow analysis: The data flow analyzer uses both the REFINE language and the pattern-matching capability. For example, one analysis step to be performed is to determine whether a particular JCL statement is a DD statement. This condition can be expressed as follows:

stmnt = '//'&oddnameDDdsn=&dssname, , disp=&displst, .'

The above expression can be used to test, or pattern-match, the value of stmnt against the program template on the right side of the equality. The match succeeds if the following conditions are met:

- stmnt is a statement
- stmnt is a DD statement
• `stmt` has a DSN and
• `stmt` has a DISP.

If the match succeeds, the `REFINE` pattern variable `DSNAME` is bound to the name of the statement, `DSNAME` is bound to the DSN parameter list, and `DISPST` is bound to the DISP parameter list. The program template is written primarily using the syntax of JCL, with the addition of the syntax for `REFINE` pattern variables (the `@` followed by the name of a `REFINE` variable).

3.3 Program Transformation Examples

These examples focus on perhaps the most novel capability of the `REFINE` system, specifying and automatically executing transformation rules that perform complex modifications to software. This is the heart of providing automation for software maintenance and re-engineering activities. The analysis activities discussed earlier are performed with the goal of determining where or how subsequent modifications to the software should be made.

3.3.1 Porting C Applications to a Microcomputer

One common instance of non-portability in programs written in portable languages is the syntax of identifiers. For example, in the C language, newer implementations for engineering workstations usually allow identifiers to be more than eight characters long, but implementations for smaller machines often require identifiers to be eight characters or less. Therefore porting a C program to run on a small machine may require renaming all identifiers longer than eight characters to be shorter than eight characters.

This transformation is simple to describe and formalize, complicated to implement correctly using a text-based approach, but easy to implement using a transformation-rule based approach. In fact, a `REFINE` program that performs this program transformation correctly is about twenty lines long and easy to understand.

It is instructive to examine some of the complications that arise in performing this transformation using a text-based approach, because they are typical of problems that arise in performing non-trivial program manipulations on text strings.

• Lexical analysis: the text-based approach requires performing the equivalent of lexical analysis to determine which character string in the input file define tokens in the C language. The lexical analyzer must parse comments and string constants correctly so that they are not mis-identified as tokens.

• Identifying identifiers: the text-based approach requires that tokens that are identifiers be known, so that keywords in the language are not inadvertently renamed.

• Avoiding conflicts: the text-based approach requires keeping track of which identifiers (both original and shortened) have been encountered so far, to avoid creating a name conflict arising from C's scoping rules.

Getting all the details correct using this approach will take time and experimentation. The program will duplicate many of the analyses performed by a C compiler during parsing, breaking the input into tokens, performing lexical scope analysis, etc. On the other hand, the `REFINE` program that performs the same task has none of these problems. The C program stored in the database embodies the results of syntactic and static semantic analysis.

Here is the `REFINE` rule that forms the heart of the required program transformation:

```plaintext
rule rename-long-identifier (id) % The input to
  identifier(id) % the rule is an object "id"
  length(name(id)) > *max-id-length* % and id's
  % name is longer than the limit
  --> % then
  new-name = make-new-name(id) % generate a
  name(id) = new-name % new name called "new-name"
  % rename the
  (ref in identifier-references(id)) % for each
  % occurrence of the id in the program
  --> name(ref) = new-name) % rename it
  % to preserve consistency.
```

The `REFINE` code that applies this transformation to an entire C program is:

```plaintext
preorder-transform(my-program, ['rename-long-identifier'])
```
which can be read as “traverse the tree rooted at the object my-program, applying the rule rename-long-
identifier at each object in the tree”.

A similar approach can be used to solve related prob-
lems that arise in software maintenance, such as the
problem of merging two large programs written in a
language that does not support any scoping mechanism
for functions.

3.3.2 Converting between incompatible ver-
sions of a programming language

Two of the advantages of high-level languages over as-
sembler and machine languages are portability and ease
of integration. In theory, porting a program written in a
high-level language to a different machine is easy if
there exists a compiler for same language on the target
machine; one simply recompiles the source code. Also,
in theory, high-level languages make it easy to combine
programs using mechanisms such as external subroutines.

In practice it is not that simple because popular high-
level languages (e.g., FORTRAN and COBOL) have a
maddening variety of dialects and versions that at least
partially (and sometimes completely) remove the ad-
vantages mentioned above. Programs written in one di-
ialect cannot be compiled with a compiler written for an-
other dialect, and programs written in different dialects
cannot necessarily call each others' routines. These in-
compatibilities between dialects introduce the need to
convert programs among dialects in order to combine
them and run them on different machines. Program
transformation is often the most effective technology for
automating these conversion processes, because the
necessary changes are structural (rather than textual)
in nature, and must be made pervasively across large
bodies of application source code.

An example of such a conversion task arises in con-
nection with the language NATURAL. NATURAL is
a COBOL-like language used primarily for MIS appli-
cations. There are several versions of NATURAL on
different machines with varying degrees of compatibil-
ity. In particular, there is a recent version (NATURAL 2.0) that runs almost identically on IBM mainframes
and DEC equipment. The older and most widely used
version for IBM mainframes is NATURAL 1.2, which is
substantially incompatible with NATURAL 2.0. Thus
customers must convert NATURAL 1.2 applications to
NATURAL 2.0 and then recompile them in order to run
them on DEC equipment.

A fully automated NATURAL 1.2 —> 2.0 converter
is currently under development using REFINE. Be-

![Figure 7: NATURAL Conversion/Re-compilation Cycle](image)

fore choosing REFINE, the customer investigated sev-
eral strategies for conversion tools, including text-based
approaches and approaches using YACC and C. The
customer and Reasoning Systems jointly developed a
prototype converter using REFINE in two weeks. The
prototype included parser/printers for subsets of both
NATURAL 1.2 and NATURAL 2.0, and transformation
rules that handled several key incompatibilities between
the two language versions. The prototype was able to
completely convert several examples.

Below is an example conversion rule used in the NATU-
RAL 1.2 —> 2.0 prototype converter. This rule makes a
conversion that is necessary because NATURAL 1.2 al-
ows variables to be defined anywhere within a program,
whereas NATURAL 2.0 requires that all variable defi-
nitions occur within a single data definition clause at
the beginning of the program.

```plaintext
rule hoist-variable-definition (node) 
  variable(node) &
  var-fmt = variable-format(node) &
  *data-def-clause* = 'DEFINE DATA LOCAL
  $old-defs END-DEFINE'

  -->

  variable-format(node) = undefined &
  new-var = make-variable(var-name, var-fmt) &
  *data-def-clause* =
  'DEFINE DATA LOCAL $old-defs, $new-var END-DEFINE'
```

In this rule, the input node is first tested to see if it is
an in-line variable declaration. If so, then var-fmt is
set to the format of the inline declaration. Then (on the
right hand side of the arrow), a new variable declaration
new-var is created and added to the “data definition”
clause *data-def-clause*. Also, the variable format of
node is erased, effectively deleting node from the pro-
gram.

4 Summary and conclusion

Software maintenance and re-engineering is charac-
terized by an emphasis on analyzing and transform-
ing existing software. This emphasis distinguishes re-
gineering from forward engineering and demands a
correspondingly different set of CASE tools.
We have described an approach to software maintenance and re-engineering based on object-oriented databases integrated with LALR parsing technology. Software is captured and represented as annotated abstract syntax trees that are stored in a database. Program specification and pattern matching capabilities are used to describe and analyze software once it is in the database. Program transformation rules automatically perform extensive modifications to the software.

We have described Refine, an environment for program representation, analysis and transformation that provides the tools needed to implement the automation of software maintenance and re-engineering. The transformational approach has been illustrated with examples taken from actual experience in re-engineering software in C, JCL and NATURAL.

The ability to support automation in modifying large software systems by using rule-based program transformation is a key innovation of our approach that distinguishes it from tools that focus only on automation of program analysis. The approach has been validated by applications to software in a wide variety of languages.

References


