Extract: An Extensible Transformation and Compiler Technology

George T. Heineman and Paul Calnan
WPI Computer Science Department
heineman@cs.wpi.edu

Abstract

There is increasing interest and a need for code transformations to help designers and programmers modify source code "at a high level". However, it is a challenge to write effective software transformations and make them available for use. We approached this problem as software engineers and developed a language and supporting infrastructure that helps users specify, execute, and reuse transformation modules; after all "Software code transformations are software too". We describe our approach, our design decisions, and evaluate the resulting Extract technology with two case studies. Our results show promise in building transformation units that can be composed together (according to a well-defined semantic model) to form more complex ones.

1. Introduction and Motivation

Many design and coding tasks can be described as *code transformations*, where a user selects a desired code modification (*the goal*), and a sequence of transformations is applied to the code base (*the process*). Code transformation consists of four phases:

1. *parse* the source code file(s) into an internal model of the code
2. *search* the model for targeted fragments that must be modified to satisfy the goal
3. *transform* the internal code model as specified by the process
4. *output* the source code file(s) to be compiled or edited within a development environment

In this paper, we describe our Extensible Transformation and Compiler Technology (Extract) project, an infrastructure that makes novel contributions to the *search* and *transform* phases while using existing technology to *parse* the source code. Two projects provided the direct inspiration to pursue the Extract research effort. Each project required a specific, narrow code transformation that was provided either by *ad hoc* means (such as scripts and small hard-coded programs) or by using existing tools that could not be easily extended to investigate alternative transformations.

1.1. Active Interface Design Environment

Active interfaces are a technology for creating black-box adaptable software components [4]. An active interface decides whether to take action when a method is called; there are two phases to all interface requests: the "before-phase" occurs before the component performs any steps towards executing the request; the "after-phase" occurs when the component has completed the request. These phases are similar to the Lisp advice facility [6].

The initial Active Interface Development Environment (AIDE) used a shell script to parse the Java source code (using JavaCC [10]) and then injected small text fragments into method bodies to create the desired functionality. The script, naturally, was complex with many special cases [7].

1.2. Run-time Interface Checker

Run-time enforcement of behavioral contracts has been studied extensively in procedural and object-oriented languages. However, component-based software engineering (CBSE) imposes additional restrictions for integrating run-time enforcement of behavioral contracts into the component model. We developed a standardized service that could be added to component model implementations to enable application assemblers to enforce local properties as specified by the components in the application as well as global properties as specified by the application [14]. Given an interface specification with `@pre` and `@post` conditions specified for each method, the Run-time Interface Specification Checker (RISC) compiler converts these pre- and post-conditions into executable checking units (ECUs). These units use assertions to verify the stated conditions.

RISC supported specifications written using a subset of OCL [8]; however the RISC system relied on complicated hand-crafted code that parsed the specifications and enforced the assertions.

2.4. Requirements

The determination to combine and generalize the solutions as posed by AIDE and RISC presented an opportunity to clearly define a code transformation research agenda.

- Grammar independence – our early efforts were limited to Java source code. We recognized the need to design transformations in any language regardless of the grammar of the source code. This challenging task could only be accomplished by reusing existing parsing and compiling techniques.
- Transformation specification – the AIDE transformations were hard-coded; the contract validation code for RISC quickly became a burden for the programmers. We need to separate the generic capabilities of *searching* through code from the *transformations* and domain-specific modifications required to change the code.
- Transformation reuse – Complex transformations can be composed from other transformations. All the lessons of software reuse apply, namely, encapsulation, information hiding, and extensibility. To borrow a phrase from Lee Osterweil, “Transformations are software too” [5].

By applying the best practices of software reuse to software transformation, we are building an infrastructure where *transformation modules* can be designed and used in “black box” fashion. Users of these modules do not need to concern themselves with the (often complex) logic that supports the transformation; rather they can use the module according to its specification and even extend it to create new meaningful transformations of their own.

2.4. Small example

We introduce the topic of transformation with a small example. As the Java language evolved to JDK 1.5, the keyword `enum` was added; naturally, those programs written prior to 1.5 may have used `enum` as a variable. One example is the auto-generated code from the JTB generator [15], which includes the following snippet in all of its generated code:

```
for (Enumeration enum = jj_expentries.elements();
     enum.hasMoreElements();)
{
  int[] oldentry =(int[])(enum.nextElement());
     ...
}
```

This Java code will fail to compile with the 1.5 `javac` compiler. A small converter program (shown in Fig. 1) might be offered as an *ad hoc* solution to rename this variable by updating the source to replace certain "`enum`" strings with "`e__m`" (unlikely to be used in the program). This approach is inadequate since a `StringBuffer` object offers no high-level support for code transformation. In addition, such temporary (and throw-away) solutions inextricably link *search* and *transform* operations, thus they can’t be composed together easily, nor can they be extended.

```
// PARSE source file
File f = new File(args[0]);
FileInputStream fis = new FileInputStream (f);
byte allIn[] = new byte[(int) f.length()];
fis.read(allIn);
fis.close();

StringBuffer sb = new StringBuffer(
                     new String(allIn));

// SEARCH for 'enum' and TRANSFORM to 'e__m'
int idx;
while ((idx = sb.indexOf(" enum ")) != -1) {
  sb.replace(idx+2,idx+4,"__");
}
while ((idx = sb.indexOf(" enum.")) != -1) {
  sb.replace(idx+2,idx+4,"__");
}
while ((idx = sb.indexOf("(enum.")) != -1) {
  sb.replace(idx+2,idx+4,"__");
}

// OUTPUT transformed file
FileOutputStream fos=new FileOutputStream (f);
fos.write(sb.toString().getBytes());
fos.close();
```

Figure 1. Ad hoc code transformation

This same effect can be achieved using the Extract module, and supporting implementation, shown in Fig 2. The Patcher module processes the Abstract Syntax Tree (AST) for the Java Grammar and searches for those `Variable` and `VariableDeclarator` typed nodes and applies a Rename transformation to change variable names to "`e__m`" as needed; these node types are derived from productions in the Java grammar.

```
module Patcher {
 // Execution Block (encodes SEARCH)
 Patcher () {
  FOREACH VariableDeclarator vd =
    "//VariableDeclarator[@Variable='enum']" {
   ApplyTransform (Rename("e__m"), vd);
  } { /** if no VariableDeclarators. **/ }

 FOREACH Variable v = "//Variable" {
   if (v.toString().equals ("enum")) {
     ApplyTransform (Rename("e__m"), v);
   }
  } { /** if no Variables. **/ }
 }

 // TRANSFORM Block
 transform Rename (String name) {
  (Variable v) {impl.rename(v, name);}
  (VariableDeclarator vd){vd.setVariable(name);}
 }
}


public class Patcher_impl implements ModuleImpl {
 void rename (Variable v, String name)
   throws ModuleException {
  try {
    v.replace ((ParseTree)
      ExtractAPI.createElement(name,"Variable"));
 } catch (ParseTreeException e) {
   throw new ModuleException (e);
  }
 }
}
```

Figure 2. Sample Extract transformation.

We now present the core design decisions and describe the semantic design and execution model for Extract. We discuss two case studies in the use of Extract and evaluate its performance and how well we meet our requirements. We compare other approaches in the field and conclude with a discussion on the extensibility of Extract and some lessons learned.

3. Design Decisions

The design decisions for Extract can be broken down by phase. The most important principle we follow is the independence of Extract from the underlying grammar of software being transformed.

3.1. Parse phase

We searched for technologies that would enable us to both parse and modify code. OpenJava is a Meta Object framework that lets one easily create and manipulate representations of java code as Abstract Syntax Tree (ASTs) [9]. For example, the code snippet in Fig. 3 creates the canonical Java "HelloWorld" program. Our early success in Extract is due to OpenJava, since it helped parse, construct, and manipulate ASTs representing Java classes.

However, we did not want Extract to be directly coupled with OpenJava. Also, constructing a Meta Object protocols (MOP) for a grammar is expensive, and one might not always exist for a desired grammar. To configure Extract to use arbitrary grammars, we turned to JavaCC the Java Compiler Compiler [10], an extraordinarily useful tool that generates a lexical analyzer and parser given a `grammar.jj` description of a grammar. A companion tool JTB (the Java Tree Builder) takes the `grammar.jj` file and generates AST classes and supporting traversal classes [15]. The grammar is fully integrated into Extract once key specialized classes are written.

```
MethodCall mc = new MethodCall(
   "System.out.println",
   new ExpressionList(
     Literal.makeLiteral ("Hello World")));
MethodDeclaration md = new MethodDeclaration(
   new ModifierList (ModifierList.PUBLIC |
                     ModifierList.STATIC),
   new TypeName("void"), "main",
   new ParameterList (new Parameter(
     new TypeName ("String", 1), "args")),
   null, new StatementList (
      new ExpressionStatement(mc)));
ClassDeclaration cd = new ClassDeclaration(
   new ModifierList(ModifierList.PUBLIC),
   "HelloWorld", null, null,
   new MemberDeclarationList(md));
```

Figure 3. Construct HelloWorld Java class

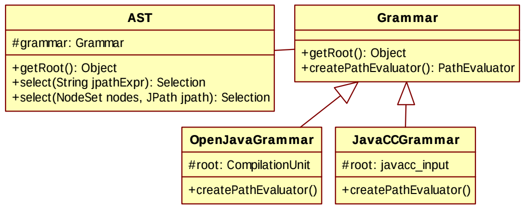

Figure 4. Internal representation

The Extract run-time parses and loads input files as a set of ASTs stored by a *container*. The AST structure is shown in Fig. 4. Extract cares only that the ASTs are associated with the relevant `Grammar` interface. For grammars provided as JavaCC `.jj` files, one can readily construct the appropriate `Grammar` subclasses.

3.2. Search phase

We decouple the *search* for nodes in the AST from the *transformations* that affect them. We designed the JPath language (inspired by XPath [1]) to traverse an

AST to locate desired nodes within it. For example, given an AST representing a Java file, the query `"//ClassDeclaration[@Modifiers contains 'public']"` returns node(s) of type `ClassDeclaration` representing the public class within the file (if it exists). The identifier `ClassDeclaration` is interpreted as the type of a node in the AST. JPath has no preconceived types and operates via reflection over the specified input grammar of the source being transformed.

A well-formed JPath query has the form "$\text{Step}_1/\text{Step}_2/\ldots/\text{Step}_n$" and is evaluated (step-by-step) over an AST given a set of *anchor* nodes (most frequently just the root node). Each step is typed to locate suitable child node(s) of the anchor node(s) that belong to the desired type; the resulting nodes found become the anchor nodes for the next step. Wildcards can be included in the first n-1 steps, such as `*` (select all children) and `//` (select all descendants), but the nth step must be fully typed. Each step may have an additional filter (in `[]` brackets) based upon the attributes of the selected node(s).

A JPath query either evaluates to a `Selection` representing the empty set ∅ or a set of typed nodes as determined by the nth step. The following example shows how one can extract the raw call graph from a set of nested JPath expressions over an AST object, `src`, representing a Java source file.

```
Selection s, s1, s2;
s = src.select("//ClassDeclaration");
Iterator i = s.iterator();
while (i.hasNext()) {
 ClassDeclaration d = (ClassDeclaration) i.next();
 s1 = src.select(d, "//MethodDeclaration");
 Iterator i2 = s1.iterator();
 while (i2.hasNext()) {
  MethodDeclaration m=(MethodDeclaration) i2.next();
  s2 = src.select(m, "//MethodCall");
  Iterator i3 = s2.iterator();
  while (i3.hasNext()) {
   MethodCall c = (MethodCall) i3.next();
   // report call graph on (d, m, c)
  }
 }
}
```

Figure 5. Nested JPath expressions

While JPath provides the raw capability of searching for desired nodes of interest within an AST, we need additional support to effectively declare and implement transformations. Towards this end, each Extract module may have a single execution block that contains embedded JPath queries. Much like the enhanced **for** loop for JDK 1.5, we designed a construct that hides the JPath details found in Fig. 5; queries can be nested, in which case the optional `[(node)]` entity serves as the anchor for the query.

```
FOREACH  [(node)]  Type  t  =  "JPathQuery"  {
  // code executed once for each node
} {
  // code executed if none found
}
```

The ExtractC compiler compiles these code fragments into Java code similar to Fig. 5.

The prime motivation behind Extract was to make it possible to specify transformations separate from the search queries. In the example from Fig. 2, the top block contains the Patcher Extract module which consists of an execution block with two queries and a transform block with two cases. As we shall see in Section 4, this separation makes it possible to extend modules and create new modules easily.

3.3. Transform phase

The transform block contains a series of named transformations, each with its own parameter list. Related transformations are grouped together according to the following syntax:

```
transform <name>(<args>) {
  (<case 1>) { /** logic. **/ }
  (<case 2>) { /** logic. **/ }
     …
}
```

To ensure that transform logic is simple, the code of the *logic* block cannot include methods that throw Exceptions. In this way, the Extract module code should be abstract enough to understand without complications. Naturally, there must be some way to include more complex code, and that specialized – often domain-specific – code, is coded in the Implementation class associated with the Module. These implementation methods become abstract concepts that can be used without exposing their complex logic. Each module instance is instantiated with a reference to its corresponding implementation object `impl`, which is available within the execution or transform blocks.

As an example, the transformation shown in Fig. 6 adds the interface `pkg.iname` to the implements clause of a class declaration. Having described all essential elements of Extract modules, we present the semantic model that explains how Extract modules are designed and executed.

```
transform AddInterface(ClassDeclaration c) {
   (String pkg, String iname) {
 TypeName[] old=c.getInterfaces();
 TypeName[] mod=new TypeName[old.length+1];
 System.arraycopy(old,0,mod,0,old.length);
 mod[old.length] = (TypeName)
   ExtractAPI.createElement (pkg+"."+iname,
                              "TypeName");
 c.setInterfaces (mod);
 }
}
```

Figure 6. Sample Transform

4 Extract semantic model

An Extract module represents an encapsulated unit of transformation. It contains an *execution block*, E, that processes a single AST and potentially transforms that AST using a set of defined *transformations*, T; these transformations `t`$_i$`(args)` are the fundamental granularity of source code transformation in Extract. Complex helper computations are coded in an *implementation class*, I, associated with each Extract module. After the completion of the execution block, a set of defined *properties*, P, is available for get/set access to support the communication between modules. Thus, an Extract module M is defined by the tuple (P, E, T, I); each of these elements is optional.

In short, the execution block of an Extract module contains embedded JPath queries over the AST to locate specific nodes that are transformed by the defined transformations whose logic is enhanced (or customized) by implementation classes; state computed by the module is retrieved via its properties.

An Extract *main* module provides an entry point for complex transformations, and acts like the `main` function in a C program. It receives as input the Extract *container* which contains the set of pre-parsed and loaded AST objects. A main module may instantiate and invoke other Extract modules on the AST trees. Once the main module completes, all ASTs managed by the container are written to disk.

With the Extract capabilities described so far, users would be able to define and use transformation modules; they could be easily assembled together to form more complex ones. However, the true power of Extract becomes realized when designers use *single inheritance* to create new modules from existing ones.

4.1. Module semantics

Module writers can use inheritance to create new Extract modules by extending an Extract module. A module M_{spec} can extend module M_{gen} to inherit (and possibly override) the core module elements of M_{gen}, in the same manner as a subclass extends an object-oriented class. Each of the core module elements is optional, resulting in different types of modules. The implementation I contains a set of Java helper methods to aid transformations.

There are three independent ways that a specialized module can extend a module; (1) Override the existing implementation class to change the underlying way in which the module carries out the transformation specified in M_{gen}; (2) Override the defined transformations to change the fundamental building blocks of transformations; (3) Override (or provide) the execution Block to select different nodes to be transformed.

When module M_{spec} = (P_{spec}, E_{spec}, T_{spec}, I_{spec}) extends module M_{gen} = (P_{gen}, E_{gen}, T_{gen}, I_{gen}), the semantic structure of M_{spec} follows the standard approach as found in Java inheritance, resulting in $M_{spec} = (P_{gen} \cup P_{spec}, E_{spec} \bullet E_{gen}, T_{spec} \bullet T_{gen}, I_{spec} \bullet I_{gen})$, where properties are unioned together and the • operator represents the replacement of the core element(s) in the generic module M_{gen} with those in M_{spec} that override them. The semantic model is enforced by the way ExtractC compiles the Extract Modules into Java code (as described in Section 4.3).

Finally, an *interface* module (P, ∅, ∅, ∅) declares an interface by defining a set of properties P, each accessible via get/set method calls. Because modules are independent from each other, their only communication should be through properties. To enable this communication, Extract modules can import any number of interface modules (this construct is analogous to interfaces in Java).

4.2. Execution Semantics

The Extract run-time provides a command-line interface to execute Extract Modules. To execute a module, the run-time needs to be told (1) the set of input source files to be transformed (these could be found on disk, or in a JAR or ZIP archive file); (2) the grammar to use while parsing the input files; (3) an output directory; and (4) the name of the Extract module to invoke. The run-time parses all appropriate source files into ASTs using the specified grammar, and instantiates an object representing the specified module. If the module is a main Module, then it receives the Extract container as input, otherwise the module is iteratively fed each AST one at a time. Once execution completes, all ASTs in the container are written to the selected output directory.

The execution of two modules can be composed together in *sequential independence* within a Main module as follows:

```
module Main {
  main (String[] p, SourceContainer c) {
    INVOKE ModuleP p() FOREACH c;
    INVOKE ModuleQ q() FOREACH c;
  }
}
```

In the above example, Module p is invoked on each AST, one at a time until all ASTs in the container have been processed. Then Module q is similarly invoked on each AST. Another alternative, which we call *sequential dependence* occurs when, for each AST in the container, the second modules is executed immediately after the first one:

```
module Main {
main (String[] p, SourceContainer c) {
  for (Iterator it = c.iterator();
       it.hasNext(); ) {
    AST src = (AST) it.next();
       INVOKE ModuleP p() ON src;
       INVOKE ModuleQ q() ON src;
     }
}
```

4.3. Compilation Semantics

The ExtractC compiler converts an Extract module source file `Module.xm` into a set of three 100% pure Java files:

- `Module_impl.java` – this implementation class is generated only once. Thereafter, the module writer can implement special functionality in this class as needed for complex transformations
- `Module_props.java` – this interface declares the set of properties supported by the module
- `Module.java` – this class contains the execution block that defines the logic of the transformations defined by the module. Transformations defined by the Extract module appear as inner static classes for use by the Execution Block.

To execute a module, one need only instantiate an object of the generated Module class and invoke its execute (AST) method. The ExtractC compiler converts transform declarations into static classes within the enclosing Extract `Module` class. When M_{spec} extends M_{gen}, the implementation class M_{spec}_impl extends M_{gen}_impl. Similarly, the inner static transform classes abide by the same extension, to enable substitutability.

5. Case Studies and Evaluation

We evaluated Extract with two considerable case studies. We describe each and then evaluate their strengths and weaknesses.

5.1. Code Obfuscation

Given a set of Java source files, we would like to obfuscate all method names to make it impossible to understand the original code. The resulting transformed code must compile and execute the same as the original. To solve this problem over a set of ASTs, we must analyze the global set of package definitions to determine the scope of the original source. Method declarations within this scope cannot be obfuscated if they (a) are the implementation of an interface external to the scope; or (b) override a method whose definition is external to the scope.

We designed three Extract modules to execute in the following order:

- `PackageRegistrator` – identifies the scope of the source ASTs by recording their package names
- `ClassInspector` – builds up symbol table information for all source code blocks so that every method call is properly typed; also identifies whether method declarations meet the obfuscation criteria above
- `Obfuscation` – Performs the requisite modifications to both method declarations as well as method invocations

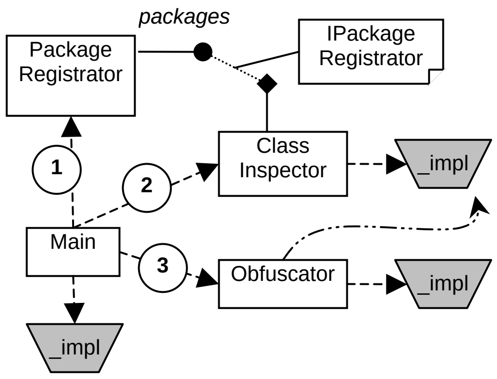

Figure 7. Obfuscation Logic

Fig. 7 contains a representation of the overall control structure. The complex details regarding symbol tables are safely encapsulated and hidden in implementation files. The communication between the `PackageRegistrator` and `ClassInspector` is governed by means of properties – the ideal means of interaction. In short, `ClassInspector` demands only that someone provide it the set of packages. However, note that `Obfuscator` makes a direct method call to the implementation of another module. This interaction is clearly not desirable because it tightly couples these two Extract modules. The communication is simply to static methods (i.e., `ClassInspector_impl.isInScopeClassMethod( className, name)`. We are currently working on extending the ability of modules to export interfaces more complex than simple get/set; this should resolve the tight coupling evident here.

Given a small GUI editor of 9,748 lines of Java code in 75 files, we ran the obfuscation process five times on an Dell Computer with an Intel Pentium 4 CPU of 2.8 Ghz and 504 MB of RAM. We eliminated the slowest and fastest execution times and the average running time for the remaining three runs are 34.3 seconds to parse the input, and 6.0 seconds to complete the obfuscation process. In total, 575 method declarations were inspected and 449 of them (78.0%) were obfuscated. A method declaration was obfuscated if it was determined to be "in the scope" of the set of input files. For example, the method `actionPerformed (ActionEvent ae)` in a class that implements `ActionListener` is considered "out of scope" because the method is defined in a package not provided in the input. Of the 3,253 method invocations, 798 of them (24.5%) were altered. This case study was an excellent "stress test" for Extract.

We also ran a side-by-side comparison with the publicly available ProGuard system, which shrinks, optimizes, and obfuscates source code [17]. On their "JDepend" test, which required 8 seconds to obfuscate a project with 22 files and 57K bytes of code, our Extract tool performed the same task in 13.3 seconds on comparable hardware (obfuscating 259/313 method declarations and 602/2203 method invocations).

5.1. Condition Checker

Our second case study provides a solution to the problem of checking the `@pre` and `@post` conditions associated with method declarations in an interface. The essential problem we solve was posed by Findler et. al, namely, how to assign blame when the pre- or post-condition for a method has been violated [16]. In their paper, the authors state "An implementation of our contract checker is in progress…. We plan to release a prototype by the end of 2001." When no such prototype became available, we developed our own case study to implement the extensive analysis described in their paper.

As we focused on the problem, we realized that the first two steps from the Obfuscation example could be used as is. Thereafter, we needed to perform three additional steps:

- `HierarchyVisitor` – captures all 'extends' and 'implements' relationships between all classes and interfaces within the input ASTs
- `MethodCallTransformer` – the solution proposed by Findler [16] is to add customized contract-checking methods within the interfaces and the code invoking the original methods. Thus some original method invocations have to be changed
- `ConditionAdder` – to produce the required set of contract checkers, we need to add several new method declarations (complete with the logic to validate the conditions) and several new classes

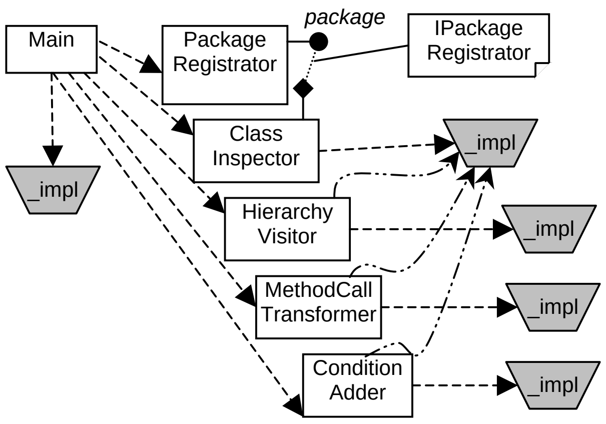

Figure 8. Contract Checking Logic

As with the obfuscation example, the excessive interaction with the implementation for `ClassInspector` reveals that we need to improve interface declarations for Extract modules. We successfully implemented contract checking and uncovered a special case not fully explored in [16].

Table 1. Statistics for Obfuscation

Obfuscation	**XM**	**#Trans**	**impl**
ClassInspector	70	0	696
IPackageRegistry	21	0	69
Main	43	0	79
Obfuscator	123	2	166
PackageRegistrator	48	0	21
Helper Classes			2291
Total	305		3324

Table 2. Statistics for Contract Checking

Contract Checker	XM	#Trans	impl
PackageRegistrator	48	0	21
ClassInspector	70	0	696
ConditionAdder	623	4	165
HierarchyVisitor	87	0	92
Main	48	0	89
MethodCallTransformer	66	1	111
Helper Classes			592
Total	942		1766

6. Related Work

There are families of approaches towards transforming source code. While space does not permit an exhaustive comparison, we try to provide sufficient coverage of the related areas.

7.1. Scripts

There is a wealth of research in optimizing programs (especially parallel ones) by means of transformations over loops and other computationally expensive operations. In many cases (i.e., [18]) after analysis reveals poor performance, the desired transformations are described by means of a script. In reality, the complex analysis is simply another process for performing a search over the input files (using domain-specific and often run-time information), and the results of the search instantiates specific instances of our transform blocks. We believe that Extract is ill suited for the fine-grained analysis and transformation of source code for performance-optimization.

7.3. XML as common representation

A common solution to dealing with different input formats is to agree on a common representation into which input files can be converted (and de-converted) without loss of information. In this way code transformations over the common representation would be made available to any grammar. Collard and Maletic propose to use XML as the standard format, and thereby enable any of the available XML transformations (i.e., XSLT [2] or XQuery [3]). The benefit of this approach is that it directly leverages the numerous tools that are immediately available for transforming documents. While it is tempting to view code transformation as strictly an exercise in document manipulation, software has complex semantic relationships between its constituent elements that doesn't conform to the hierarchical structure of an XML document. Also, from our perspective, Extract modules can build state during the transformation process and one can quickly augment the transformations with domain-specific helper methods written in the module implementations; these capabilities are not available in XSLT. Also, while XSLTs can be viewed as modules, it is very challenging to compose them together into more complex transformations.

7.4. Algebraic Transformation

Researchers have used relational algebras and relational calculators (such as Grok [11]) to specify both low- and high-level code transformations [12]. Fahmy et. al construct a graph-based model of a system where nodes represent subsystems and modules and edges represent relationships between the nodes (such as contains, uses). With the help of a series of relational calculations in Grok script, they perform transformations to better understand the architecture and its dependencies. Lin and Holt use Grok to extract information about software [13]. For example, to create the call graph for a program, one must: 1) find each function definition; 2) find each function call; and 3) determine which function definition contains which function call. Using Grok, this would be encoded as:

```
nd := $INSTANCE . function_decl
fd := nd o body
n2 := $INSTANCE . call_expr
fc := n2 o fn o op0
call := fd o (contain +) o fc
```

Here, `fd` is computed to be the set of the code bodies of all function declarations. `fc` is computed to be the set of all *called functions*. Starting from those nodes representing call expressions, traverse the function (fn) edge to find the node representing the function, then traverse the `op0` edge to locate the actual function declaration. (Note the similarity in structure to the example in Fig. 5, which, by the way, contains the Extract implementation of this call graph problem). The call graph is then computed as the relation between a function declaration and a function call via one or more edges from the contain relation. Algebraic transformation specifications are concise and unambiguous and eminently reusable, but their weakness is the inability to augment the transformations with meaningful helper methods.

7.5. IDE support

Modern integrated development environments help support programmers in numerous ways. Most recently, code editors have begun to provide options to help programmers make source code modifications “at a high level” rather than forcing the user to accomplish all programming by simply text-based modification.

Eclipse has extensive support for refactoring using an Application Programmer Interface (API) for semantic preserving workspace transformations [20]. There is no space here to present the details of the classes that manage the refactoring choices presented to the user. However, Eclipse has classes for each refactoring task (in the package `org.eclipse.jdt.internal.corext.refactoring.rename`, for example). In each case, the refactoring class responsible for the respective task (i.e., rename a field, rename a method) contains the search logic, the transformation logic, and all required error checks. Such functional encapsulation fails to adequately reuse code or enable transformations to be effectively composed together without complex wizards.

8. Lessons learned and future work

Our experience in developing Extract and the corresponding cases studies has highlighted some important lessons. First, there is a need for more powerful ASTs. In our case studies, the most complex (and tedious to write) modules were the ones that initially traversed the Abstract Syntax Tree to build up semantic information required by future transformations. Ideally, most Extract users would not have to design these, since once written for a grammar, they could be reused as is to enhance productivity. Second, while our focus has been (and will continue to remain) code transformation, there are many opportunities to use the Extract infrastructure for analysis by mining the source code for relevant information. Third, as Extract has grown and expanded, we have witnessed a bootstrapping process, which further solidifies the contributions of Extract. Our earliest version relied solely on transformations enabled by OpenJava [9]. As we re-engineered Extract to be grammar-independent, we wrote several Extract modules whose sole purpose was to “transform” code generated by JTB [15] into the key configuration classes hinted at in Section 3.1.

Extract version 2.0 is freely available [21], as both a standalone infrastructure (built using the `ant` tool and executable via command-line scripts) and an Eclipse Project. Work on integrating Extract into Eclipse continues; our goal is to add a menu item under the Refactor menu to enable end-users to select pre-designed Extract modules to transform their code, as well as to enable users to select Extract modules of their own to execute. We understand that it is dangerous to allow users to write arbitrary transformation scripts that may possible irreparably damage their code, yet we must investigate the ways that ordinary programming tasks can be converted into module-based code transformations.

References

[1] W3C, “XML Path Language (XPath): Version 1.0”, http://www.w3.org/TR/xpath, Nov. 1999.

[2] W3C, “XSL Transformations (XSLT): Version 1.0”, http://www.w3.org/TR/xslt, Nov. 1999.

[3] W3C, “XQuery 1.0: An XML Query Language”, http://www.w3.org/TR/xquery, Jun. 2006.

[4] G. Heineman, “A model for designing adaptable software components”, Proceedings, 22nd International Conference on Computer Software and Applications Conference (COMPSAC), Vienna, Austria, Aug. 1998, pp. 121—127.

[5] L. Osterweil, “Software processes are software too”, Proceedings, 9th International Conference on Software Engineering, Mar. 1987, Monterey, California, p.2-13.

[6] GNU Emacs Lisp Reference Manual, Chapter 17: Advising Emacs Lisp Functions, www.gnu.org/manual/elisp

[7] Paul Calnan, “Extract: Extensible Transformation and Compiler Technology”, MS Thesis, WPI Computer Science Department, April 2003.

[8] J. Warmer and A. Kleppe, *The Object Constraint language: Precise Modelling with UML*, Object Technology Series, Addison-Wesley, 1999.

[9] M. Tatsubori, S. Chiba, M. Killijian, K. Itano, “OpenJava: A Class-Based Macro System for Java”, Lecture Notes in Computer Science 1826, Reflection and Software Engineering, Walter Cazzola, Robert J. Stroud, Francesco Tisato (Eds.), Springer-Verlag, 2000, pp.117-133.

[10] “Java Compiler Compiler (JavaCC) - The Java Parser Generator”, https://javacc.dev.java.net/

[11] R. Holt. “Structural Manipulations of Software Architecture Using Tarski Relational Algebra,”, Proceedings of the 5th Working Conference on Reverse Engineering 1998, Honolulu, Hawaii, Oct. 1998.

[12] H. Fahmy, R. Holt, J. Cordy, "Wins and Losses of Algebraic Transformations of Software Architectures", 16th international Conference on Automated Software Engineering, San Diego, California, Nov. 2001

[13] Y. Lin and R. Holt, "Software Factbase Extraction as Algebraic Transformations: FEAT", 1st International Workshop on Software Evolution Transformations, Nov. 2004, pp. 21–24.

[14] G. T. Heineman, "Integrating Interface Assertion Checkers into Component Models", Proceedings, 6th International Component-Based Software Engineering (CBSE) Workshop, Electronic proceedings, Portland, Oregon, May 2003.

[15] Jens Palsberg, Java Tree Builder, http://compilers.cs.ucla.edu/jtb.

[16] R. Findler, M. Latendresse, and M. Felleisen, "Behavioral Contracts and Behavioral Subtyping", Proceedings, Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Vienna, Austria, Sep. 2001, pp. 229–236.

[17] ProGuard, http://proguard.sourceforge.net

[18] A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam et N. Vasilache, "Facilitating the search for compositions of program transformations", ACM International Conference on Supercomputing (ICS). Boston, MA, June 2005.

[19] M. Collard and J. Maletic, "Document-Oriented Source Code Transformation Using XML", 1st International Workshop on Software Evolution Transformations, Nov. 2004, pp. 21–24.

[20] The Eclipse Foundation, http://www.eclipse.org

[21] Extract Project Home Page, https://sourceforge.wpi.edu/sf/projects/extract