Position Papers from

Software Transformation Systems Workshop 2004

STS'04

Part of Generic Programming and Component Engineering Conference 2004

(GPCE'04)

Vancouver, Canada
October 24, 2004

http://www.program-transformation.org/Sts/STS04

Magne Haveraaen[1]

Jim Cordy[2] Jan Heering[3] Ganesh Sittampalam[4]

Abstract

This is a collection of the position papers presented at the Software Transformation Systems Workshop 2004 (STS'04).

Generative software techniques typically transform components or codefragments, instantiate patterns etc. in some way or another to generate new code fragments, components or programs. Often this needs software support beyond that of existing compilers, i.e., some kind of system which takes software as inputs and produces software as output.

Software transformation systems are tools which are built for such transformations. They range from specific tools for one purpose, via simple pattern matching systems, to general transformation systems which are easily programmed to do any reasonable transformation. Thus the more general tools may be treated as meta-tools for generative programming.

This workshop is designed to investigate the use of software transformation tools as tools to support generative programming. We want to look at various generative techniques and suggest how these may be supported by various general purpose transformation tools. This may lead to a more general understanding of common principles for supporting generative methods.

[1]Department of Informatics, University of Bergen, Norway; Department of Computer Science, University of Wales Swansea, UK; http://www.ii.uib.no/~magne

[2]School of Computing, Queen's University, Canada; http://www.cs.queensu.ca/~cordy/

[3]CWI, Amsterdam, Netherlands; http://www.cwi.nl/~jan

[4]Oxford University Computing Laboratory, UK

Contents

The overview below lists the participating author, the title of the position paper, and an e-mail contact address.

Ira D. Baxter: Program Generation and Modification using Multiple Domains

Paulo Henrique Monteiro Borba: General-purpose Transformation System for Java

Marat Boshernitsan: A Case for Interactive Source-to-Source Transformations

Thomas Cleenewerck: Invasive Composition by Transformation Systems

Jim Cordy: Metaprogram Implementation by Second Order Source Transformation

Anthony Cox: Lexical Source-Code Transformation

Magne Haveraaen: Software Transformations supporting Software Engineering

Jan Heering: Generic Software Transformations

Görel Hedin: Towards comparing Transformation Systems and Formalisms

Karl Trygve Kalleberg: Tracing Abstractions throough Generation

Terence Parr: The Role of Template Engines in Source Translation Systems

Marcelo Sant'Anna: Transformation Circuits: Exploring new Paradigms for Software Transformation Systems

Shane Sendall: Understanding Model Transformation by Classification and Formalization

Ganesh Sittampalam: Extending Languages with Transformation and Generative Technology

Tony Sloane: Cooperating Generators

Douglas R. Smith: Software Transformation Systems

L. Robert Varney: Generative Programming, Interface-Oriented Programming and Source Transformation Systems

J.J. Vinju: Generation by Transformation in ASF+SDF

Eelco Visser: Reusable and Adaptive Strategies for Generative Programming

Hironori Washizaki: A Technique of Transforming parts of Object-Oriented Class Library into Structurally Reusable Components, and Its Application

David S. Wile: Transformation Systems for DSLs, Architectural Styles, and Graphical Languages

Eric Van Wyk: Semantiv Analysis in Software Transformation

For more details see the individual position papers or `http://www.program-transformation.org/Sts/STS04`

Program Generation and Modification using Multiple Domains

Ira D. Baxter, Ph.D.
idbaxter@semdesigns.com
Semantic Designs Inc.
12636 Research Blvd. #C214
Austin, Texas, USA 78759-2200
512-250-1018

ABSTRACT

Generative programming is generally discussed with respect to one target programming language. Little is said about the realistic possibility that such generation might require multiple languages in, multiple languages in intermediate stages of generation, or multiple languages out, or that these might all be different languages. However, having multiple specification languages enables one to express each concern in an appropriate notation. Having multiple intermediate languages can simplify the staging necessary for code generation. Having multiple result languages is a necessity for generation of many kinds of complex software.

This paper sketches DMS, a source-to-source program transformation tool capable of handling multiple languages at all three levels. This enables DMS to be used for a wide variety of program generation and change activities.

General Terms

Algorithms, Management, Design, Economics, Languages

Keywords

Software transformation, software analysis, migration, component architectures, legacy systems, C++, compilers, re-engineering, abstract syntax trees, patterns, rewrite rules.

1. The DMS[1] Software Re-engineering Toolkit

DMS provides infrastructure for software transformation based on deep semantic understanding of programs. Programs are internalized via DMS-generated parsers, available for most mainstream languages, and definable for others. Analyses and manipulations are performed on abstract syntax tree (AST) representations of the programs, and transformed programs are printed with prettyprinters for the appropriate languages.

The Toolkit is capable of defining multiple, arbitrary specification and implementation languages (domains) and can apply analyses and transformations to source code written in any combination of defined domains. Transformations may be either written as procedural code or expressed as source-to-source rewrite rules in an enriched syntax for the defined domains. Rewrite rules may be optionally qualified by arbitrary semantic conditions.

The DMS Toolkit can be considered as extremely generalized compiler technology. It presently provides these facilities:

- A hypergraph foundation for capturing program representations (e.g., ASTs, flow graphs, etc.).
- Complete procedural interfaces for processing ASTs, etc.
- Means for defining language syntax, and deriving full context-free parsers and prettyprinters to/from DMS internal ASTs.
- Support for defining and updating arbitrary symbol tables holding name/type/location information.
- An attribute evaluation system for encoding arbitrary analyses over ASTs. One use is constructing symbol tables.
- An AST-to-AST rewriting engine that understands algebraic properties (e.g., associativity and commutativity).
- The ability to specify and apply syntax-specific source-to-source program transformations. Such transforms can operate within a language or across language boundaries.
- A procedural framework for connecting these pieces and adding arbitrary code.
- A scalable computational foundation in the parallel language PARLANSE, DMS's implementation language.

One of the many domains implemented for DMS is C++, which has a preprocessor, parser, prettyprinter and full symbol table construction/access/update for both the ANSI and Visual C++ 6.0 dialects. Unlike a compiler preprocessor, the DMS C++ preprocessor preserves both the original form and expanded manifestation of the directives within the AST so that programs can be manipulated, transformed, and printed preserving preprocessor directives in the presence of preprocessor conditionals. DMS has similar front ends for Java and COBOL.

DMS has been used for a variety of large scale commercial activities, including cross-platform migrations, domain-specific code generation, and construction of a variety of conventional software engineering tools for dead and clone code elimination, test code coverage, source browsing, and static metrics analysis.

A more complete discussion of DMS is presented in [1]. DMS-based tools are described on the Semantic Designs web page [2].

The domain notion stems from the Draco work [4]. Domains are independent notations that cannot be confused, and are often manipulated in one tool execution instance. This is accomplished

---

[1] DMS is a registered trademark of Semantic Designs Inc.

by simply tagging every node with its domain. One can construct mixed trees, and can specify such trees in source-patterns using domain escapes. Draco however offers only source-to-source program transformations and extremely simple transformation strategies. DMS follows Draco, but goes considerably further in offering mode-based lexers and full-context free (GLR) parsers, making it practical to define arbitrary domains such as C++, and providing a fusion of source-to-source program transformations and compiler-like infrastructure such as procedural code and transformations, and explicit symbol tables as well. We believe this fusion contributes greatly to the utility of DMS.

TXL [5], ASF-SDF [6] and Stratego [7] also offer source-to-source transformation capabilities. Our understanding is they do so by defining one language to be manipulated in a tool execution instance. Processing multiple languages requires the construction of a union-grammar containing the set of BNF rules for all. This appears to make processing such union languages difficult because of the possibility of confusing a construct in one language using constructs from another (consider expressions as one example; consider translating between two dialects whose syntax is identical but whose semantics differ). This would appear to confound semantics and therefore correct transformation. Secondly, these tools all seemingly share a philosophy of avoiding building full symbol tables for the languages they process, often by encoding extra syntax (TXL, ASF-SDF) or by generating scope-specific rules (Stratego). We contend extra syntax just compounds the first problem; incompleteness makes the domain less useful. We believe that scope-specific rules confound definition with purpose; for C++, one must encode transformation rules that achieve some desired effect in conjunction with understanding the C++ scope semantics. The complexity of real languages makes this impractical.

TXL offers rule-based sequencing strategies. ASF-SDF suggests chaining many, many exhaustive transformation-set applications. Stratego has a sophisticated language for sequencing transformations by controlling the AST walk. DMS offers arbitrary procedural code, which we have found very practical.

2. Multiple domains in Practice

DMS has been used for a number of applications in which multiple domains were valuable.

The first application is DMS itself. DMS provides a number of separate domains for many of its facilities. DMS uses a domain to process its source-to-source transformation rules, concurrently (both in figurative and actual sense) with processing the other domains involved in whatever defines the current task. A second domain allows one to encode possible side effects of rules, to enable parallel execution safety analysis.

For Rockwell Automation, DMS was used to construct a factory control code generator, involving 4 domains. The input specification language was an XML representation of a graphical formalism specifying controllers of individual devices and their compound supercontrollers. The generator produced code for two relatively different dialects, with similar syntax, of Relay Ladder Logic (RLL), a kind of assembly language involving boolean arithmetic and side-effecting operations. These RLLs were semantically ugly because of their ad hoc implementation. The graphical spec was transformed into an intermediate domain, Abstract RLL, which looked nothing like either RLL but had simple, clean semantics; this enabled us to carry out strong Boolean equational simplification before transforming to the target RLLs in which this was more difficult. Having separate domain definitions for all of these meant they evolved separately, and there was no confusion of the two target RLLs syntaxes.

We are presently working on a DMS-based tool, BMT, for Boeing [3] to carry out C++ component restructuring, for a component-based embedded avionics software system having 6000+ components. The task is to convert from a proprietary distributed component architecture to CORBA/RT. One subtask requires sorting code chunks, by intended function, into different facets (interface classes) than exist in the legacy component architecture. Determining code functionality requires human understanding. To provide a clean specification facility for the Boeing engineers using the BMT, we developed a simple facet specification language. For each component, an engineer simply names the new facets and uniquely identifies which legacy methods (via simple name, qualified name, or signature if necessary) comprise its interface. The facet language itself is defined as a DMS domain, enabling easy parsing by DMS. A DMS-based attribute evaluator over the facet domain traverses the facet specifications' ASTs and assembles a database of facts for use during component transformation. Finding named code requires a full, correct C++ symbol table across the 150,000 lines of code that typically contribute to a component.

Last, carrying out software migrations typically requires multiple languages, e.g,. COBOL, SQL and JCL all at once. We have no further room to elaborate on this here.

3. REFERENCES.

[1] Baxter, I. D., Pidgeon, C., Mehlich, M., DMS: Program Transformations for Practical Scalable Software Evolution. *Proceedings of the 26th International Conference on Software Engineering,* 2004, IEEE.

[2] Semantic Designs, Inc. web site, www.semanticdesigns.com.

[3] Akers, R., Baxter, I. Mehlich, M. Re-Engineering C++ Components Via Automatic Program Transformation, *Proceedings Partial Evaluation and Program Manipulation 2004*, IEEE, to appear.

[4] Neighbors, J., Draco: A Method for Engineering Reusable Software Systems, *Software Reusability*, 1989, ACM Press.

[5] Cordy, J. TXL – A Language for Programming Language Tools and Applications, *Proc. 4th International Workshop on Language Descriptions, Tools and Applications*. ACM.

[6] Brand, M.G.J. van den, P.E. Moreau, and J.J. Vinju. Environments for Term Rewriting Engines for Free!. *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*. Springer-Verlag

[7] Visser, E., Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9, *Domain-Specific Program Generation*, LNCS 2003, Springer-Verlag, to appear.

A General-purpose Transformation System for Java

Gustavo Santos, Paulo Borba, Adeline Sousa
Informatics Center, Federal University of Pernambuco
{gas,phmb,adss}@cin.ufpe.br

Introduction

Program transformation can be considered as a unifying concept for code generation and refactoring. A refactoring comprises several behavior preserving changes on the program, but does not add new functionalities [3]. A code generation tool, on the other hand, introduces new functionalities. With this unifying view, transformations create new types and modify old ones as long as it preserves the semantics of the original program.

Most code generation and refactoring tools implement only a fixed set of transformations. This is quite restrictive because new refactorings and code patterns are often proposed, and some might be strongly related to specific design patterns, frameworks, and middleware.

In order to avoid this limitation, we are developing JaTS (Java Transformation System) a language and execution engine for defining and applying transformations. Using this language, users are able to define new transformations by not only composing existing ones but also by declaring preconditions and source and target templates that describe the code changes required by a transformation. This language's syntax is basically an extension of Java syntax with meta-programming constructs such as meta-variables, which are used as placeholders in templates. Besides meta-variables, the language provides more powerful meta-programming constructs [1,2] such as optional, conditional and iterative constructs. The language also has executable declarations that can have access to lower level code structures (syntactic trees elements) when necessary.

Language-specific versus general transformation tools

Many program transformation tools are not language-specific, being able to transform programs from an arbitrary encoded source language to an arbitrary destination language. Although this may be an advantage, it complicates the use of the tools, since the language in which the transformations are encoded is substantially different from the one to which they are applied.

There are also language-specific tools for program transformation. Most of these have the limitation of supporting only a fixed set of transformations. Some tools offer the possibility of extension through the use of an API, but this demands the user to access the system source code and actually implement the transformations using a programming language.

JaTS avoids the drawbacks of both general purpose and limited language-specific transformation tools. As JaTS's syntax is an extension of Java's syntax, it is easier for Java programmers to specify the transformations they wish to apply. Also, the transformation language takes the semantics of Java into account, for example adopting associative-commutative matching (for field and method declarations), which allows concise implementation of transformations that could be much more complicated to implement if only the syntax was taken into account.

Transformations in JaTS

JaTS transformations consist of three parts: preconditions, source and target templates. The templates consist of one or more type (class or interface) declarations. The type declarations in the source templates are matched with the source Java type declarations to be transformed; this implies that both must have similar syntactic structures. The target templates define the general structure of the types that will be produced by the transformation.

The following example shows a simple JaTS transformation that introduces a new field declaration in an arbitrary class. The construct “#Class_name” represents a typical JaTS variable. It will match with the real class name in matching process. The delimiters “#[” and “]#” indicate an optional matching. This means that the template shown in the example cans match successfully with a class declaring or not an extends clause. The variables “#Fds” and “#Mds” express the JaTS

semantic power. They can accomplish all field declarations and method declarations in the class respectively.

Source template

```
public class #Class_name
       #[ extends #Super_class ]#
{
    FieldDeclarationSet: #Fds;
    MethodDeclarationSet: #Mds;
}
```

Target template

```
public class #Class_name
       #[ extends #Super_class ]#
{
    private int newField;
    FieldDeclarationSet: #Fds;
    MethodDeclarationSet: #Mds;
}
```

Source class

```
public class ExampleClass
{
    private String oldField;
    public void method() {
    }
}
```

Transformed class

```
public class ExampleClass
{
    private int newField;
    private String oldField;
    public void method() {
    }
}
```

The application of JaTS transformations is basically based on matching, replacement and processing. The matching retrieves the information from the source Java types. There is also the possibility of passing parameters to the transformation when the matching process is not enough to get all needed information. The replacement transverses the parse-tree and replaces occurrences of variables by the values mapped to them, and the processing is responsible for the evaluation of executable and iterative meta-programming declarations.

Applicability

JaTS is being used as the transformation engine inside Coder [4], a wizard-based tool that can be used for generation and maintenance of Java programs. This experience has shown that JaTS can be used successfully for several pattern language generations, like a data collection architectural pattern for EJB, Struts-based presentation layer and others. The generated code is fully executable and requires few adjustments to be completely functional. This reduces significantly the software development cost for Coder users.

Our experience developing JaTS has shown that the system can be used in a bootstrapping way, such that the tool can support a product-line of language-specific transformation systems. Thus, it is possible to derive new transformation systems in a relative inexpensive way. Such technique is possible because JaTS transformation system model can be used for any language due to the concept of matching, replacement and processing and the adoption of the visitors design pattern.

Nowadays the JaTS transformation model includes the advantages of a language-specific transformation system, because it is easy for a programmer to learn the transformation language, and enables the user to obtain the advantages of general transformation systems, because it is inexpensive to derive a new transformation system for another language.

References

[1] Fernando Castor and Paulo Borba. A language for specifying Java transformations. 5th Brazilian Symposium on Programming Languages, pages 236-251. May 2001, Curitiba, Brazil. Also presented at the Dagsthul Seminar on Program Analysis for Object-Oriented Evolution. Dagsthul, February 2003.

[2] Marcelo d'Amorim, Clóvis Nogueira, Gustavo Santos, Adeline Souza and Paulo Borba. Integrating Code Generation and Refactoring. Workshop on Generative Programming, ECOOP'02. Málaga, June 2002.

[3] Martin Fowler et al. Refactoring: Improving the Design of Existing Code. Addison Wesley. November 1999.

[4] Qualiti Coder Tool. http://coder.qualiti.com.

A Case for Interactive Source-to-Source Transformations

Marat Boshernitsan, Susan L. Graham
University of California, Berkeley
Computer Science Division, EECS
Berkeley, CA 94720-1776
+1 510 642 4611
{maratb,graham}@cs.berkeley.edu

ABSTRACT

Many advances have been made in off-line generative and restructuring tools and in online systems for program development by refinement. However, manual large-scale modification or generation of source code continues to be tedious and error-prone. Integrating scriptable source-to-source program transformations into development environments will assist developers with this overwhelming task. We discuss various usability issues of bringing such *ad-hoc* transformations to end-users, and describe a developer-oriented interactive transformation tool for Java that we are building.

1. INTRODUCTION

Source-to-source transformations have many uses, ranging from efficiency-improving transformations in optimizing compilers, to code generation in aspect-oriented systems, to large-scale restructuring transformations such as Y2K and Euro conversions. However, one application where source-to-source transformations have seen limited use is the automation of mundane code editing tasks faced by software developers in their day-to-day work.

Changing software source code can be tedious and error-prone. The process is complicated because a conceptually simple change may entail pervasive large-scale modifications to a large portion of source code. Examples of such changes abound in the many maintenance tasks faced by developers. For instance, consider a simple task of inserting the name of the enclosing function into the code that prints an execution trace to the console. Unless the programming language provides access to "current function name" at every trace site, the implementation of this trivial change might take hours of the developer's time.

Various proposals have been made for automating systematic modification to source code. However, few tools have found their way to the "programming trenches." If a modification is a simple behavior-preserving refactoring transformation that happens to be implemented in the programmer's development environment, then the change process is quick and convenient. However, many modifications simply cannot be broken down into a sequence of well-behaved refactorings. Another option is to employ a general source-to-source transformation engine such as REFINE [1] or TXL [2]. However, specifying transformations with these tools requires familiarity with a fairly complex transformation language, so using such a system for simple changes is overkill. Alternatively, if the modification is sufficiently simple, developers may choose to use regular expression-based pattern matching facilities of Perl, SED, or other text-oriented tool. Needless to say, using regular expressions for anything but the most trivial of transformations is usually an exercise in frustration.

In this paper we argue that developers need tools for *interactive ad-hoc* transformation of source code during authoring and editing phases of software development. Transformations can be construed broadly. In addition to replacing existing code, transformations can also generate new code fragments based on linguistic structure or on meta-information embedded in program source code. In all cases, such tools must meet unique challenges posed by their interactive mode of use. Not only must interactive transformations tools be sufficiently powerful to deal with a broad range of code changing tasks, but also they must address usability issues that arise when attempting to manipulate a non-textual linguistic representation of program source code.

When thinking about and discussing software changes, developers utilize high-level linguistic structure and programming language semantics. In contrast, the developers are forced to interact with computing systems to create and modify source code using low-level text editors and representations designed for compiler input. We believe that enabling the programmers to express operations on program source code at a level above text-oriented editing will improve programmer's efficiency and result in fewer errors.

2. IF WE BUILD IT, WILL THEY COME?

To our knowledge, there are no empirical studies that suggest that programmers would use scriptable code changing tools for their everyday editing tasks. However, anecdotal evidence is plentiful. For instance, CodeGuru.com, a community website for Windows developers, includes, among other things, a list of member-contributed macros for the Microsoft Visual Studio development environment. Among the macros are: *ConvertStarComments* (replaces single-line '`/*`' comments with '`//`' comments), *InvertAssignment* (converts '`a = b`' to '`b = a`'), *DefineMethod* (automatically generates method prototypes), and others.

The willingness of developers to create such macros reflects their belief that writing a macro is more time-efficient than making the edit "by-hand," especially if the use of a macro replaces a repetitive manual action. Despite the fact that the Visual Studio macro language offers only limited text-based access source code, their utility should not be underestimated. In fact, one of the first comments on the *DefineMethod* macro in the discussion forum reads: "Cool, but I would need the reverse."

3. THE HUMAN FACTOR

Many existing tools support specification and execution of transformations on program source code. In addition to aforementioned REFINE and TXL, notable examples include TAWK [5], Inject/J [4], and the IP environment [7]. However, these tools are intended for expert use on large and complex tasks. By contrast, our system is oriented toward *end-programmers* – the

end-users of traditional development environments. We draw this distinction to differentiate end-programmers from *language tool experts*. Language tool experts are those who understand the structure of program source code from the perspective of tools, such as compilers and may be comfortable thinking about source code in terms of linguistic data structures. We do not expect end-programmers to possess this knowledge.

Nevertheless, end-programmers' understanding of program source is based on its structure. This is supported both by our empirical observations of developer expression and by the experimental results in psychology of programming [3]. When describing source code to one another, programmers say things like:

> "Put *p:= link(p)* into the loop of *show_token_list*, so that it doesn't loop forever." [6]
>
> "Change *BI_** macros to *BYTE_** for increased clarity." [8]

Programmers evoke notions such as variables, expressions, statements, loops, and assignments. They directly refer to names found in source code. They use patterns to describe large classes of similar changes. Inspired by these kinds of examples, we can design a formal language for source code transformations.

4. INTERACTIVE TRANSFORMATIONS

Guided by the above principles, the Harmonia Project at UC Berkeley is currently building an end-programmer-oriented interactive tool for source code transformation. In order to enable developers to describe transformations using familiar concepts, we targeted our notation toward the Java programming language. Our transformation language is called iXj, for "Interactive Transformations for Java." While iXj is a language tightly coupled with Java, we expect that our design methodology will be applicable for other programming languages.

Prior to designing iXj, we conducted an informal user experiment to understand what programming paradigm is most "natural" for expressing transformations. In this experiment the participants were shown "before" and "after" snapshots of a piece of source code and were asked to write down the transformation that was used to perform the change. In particular, we were interested how developers reference code fragments to be transformed, how they describe the output, and what programming style they use. We learned that to describe a location in the source code developers use language concepts ("in class *Employee*, method *getName*…") interspersed with code fragments in Java ("replace *System.out.println(x)* with…"). We also discovered that imperative programming style ("first do this, then do that") is most natural for describing modifications.

Armed with this knowledge, we based the first version of iXj on the selection/action programming model. A *selection* is a pattern that describes a set of Java source code fragments. One or more *actions* describe a transforming operation for each selection.

In order to provide scaffolding to help developers learn and use an unfamiliar notation, the iXj programs are created and executed in an integrated transformation environment. In addition to offering context-sensitive assistance during creation of iXj programs, the transformation environment enables the programmers to view partial results and to visualize execution of iXj selections and actions. Additionally, the developer can examine each transformation site, selectively undo or modify individual transformations, etc. The transformation environment can also capture the source code change history in terms of high-level transforming operations. Such a capability helps to document important aspects of program evolution, as well as supports selective rollback of high-level changes days, months, and even years after they had been performed.

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

We believe that iXj will provide the right high-level vernacular for describing code, and we expect professional developers to have no trouble specifying the control structure of pattern matching and transformations in a textual notation. At the same time, the transformation environment augments iXj with direct manipulation. Selection patterns can be created "by-example," whereby the user selects a source fragment that represented a single matching instance and then abstracts the generated pattern to match a larger class of code fragments.

5. CONCLUSION

This position paper argues for bringing interactive scriptable source-to-source program transformations into the hands of developers. A flexible end-programmer-friendly transformation tool can be used for both for systematic modification and for systematic generation of boilerplate code. As a proof of concept, we are building a novel environment that lets the programmers express operations on program source code at a level above text-oriented editing using source-to-source program transformations.

6. REFERENCES

[1] S. Burson, G. B. Kotik, and L. Z. Markosian. A program transformation approach to automating software reengineering. In *Proceedings of the 14th Annual International Computer Software and Applications Conference*. IEEE Computer Society Press, 1990.

[2] J. R. Cordy, C. D. Halpern, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. In *Proceeding of the International Conference of Computer Languages*, pages 280-285, Miami, FL, Oct. 1988.

[3] F. Detienne. *Software Design – Cognitive Aspects.* Springer-Verlag, New York, NY, 2001.

[4] T. Genssler and V. Kuttruff. Source-to-source transformations in the large. In *Proceedings of Joint Modular Language Conference (JMLC) 2003*.

[5] W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proceedings of the 4th Workshop on Program Comprehension.* IEEE Computer Society Press, 1996.

[6] D. E. Knuth. The errors of TeX. *Software – Practice and Experience*, 19(7):607-685. July 1989.

[7] C. Simonyi. The death of computer languages, the birth of intentional programming. Technical Report MSR-TR-95-52, Microsoft Research (MSR), Sept. 1995

[8] B. Wing. XEmacs ChangeLog entry for 2002-05-05. http://cvs.xemacs.org/viewcvs.cgi/XEmacs/xemacs-20/src/ChangeLog

Invasive Composition By Transformation Systems

Thomas Cleenewerck

June 30, 2004

1 Introduction

The most important strategy to deal with complex systems in computer science is the divide and conquer design paradigm. It works by recursively breaking down a problem into sub-problems until they become simple enough to be solved directly. The solutions to the sub-problems are then composed to give a solution for the whole problem. There are two kinds of composition: non-invasive and invasive composition. The non-invasive composition mechanisms are applicable as long as the kind of components to be composed fit in the dominate decomposition. However it has become clear that there are multiple equally valid and useful decompositions of the same software. In order words, there are often components that do fit and violate the dominate decomposition. There are two ways of dealing with this problem. One approach is to express a software system as a set of multi-dimensional concerns like HyperSpace [OT00]. Another approach is to keep a single dominate decomposition and express the components that violate this decomposition in a crosscutting way like AspectJ. In this later approach such crosscutting components must be invasively composed with the other components.

Quite a lot of the generative programming techniques have been build with the second approach in mind and thus offer various invasive composition mechanisms. Let us briefly discuss the most significant ones. The founder of invasive composition is subject-oriented programming [HO93]. In this model object-oriented code snippets and fragments are composed with one another using correspondence and combination rules. Gray box component models integrate [TG97] through a partial exposure of the internals of the system in terms of an operational model [BW97]. Glass-box composition models use declarative specifications to compose and reason about the composition of components [Bat03]. More recently, aspect-oriented programming (HyperJ and AspectJ) broadened the application of a crosscutting concern to a set of crosscutting points scattered over the entire software system where existing code gets composed with the crosscutting code. In fact, *every concern-specific language* ranging from general purpose languages like the ones discussed above to domain-specific languages enabling the specification of their problem into a more appropriate concern needs invasive composition mechanisms to compose these concerns. Note that the invasive composition mechanisms must not always be visible to the

developer. In the case of concern-specific languages, the more domain specific the less visible the invasive composition mechanisms will be. In short, invasive compositions are frequently needed and encountered.

2 Position

Invasive composition mechanisms are unfortunately enough still implemented with ad-hoc generators. Hereby losing valuable research results of the three main-stream general purpose transformation paradigms (GPTP): template or rule-based transformations and attribute grammars. *Our position statement is that the reason for the use of ad-hoc generators lies in the fundamental underlying in-place substitution property of those general purpose transformation.*

Template, rule-based transformations and attribute grammars are all based on an *in-place substitution* mechanism: Templates are parameterized target language expressions with escaping variables referring to any kind of domain information necessary. The templates are composed (usually by concatenation) with one another to form the whole solution. In rule based systems, the target language expression produced by rules are substituted with their top-level or pivot nodes. When no more rules apply the complete target language expression is reached. In attribute grammars attributes are attached to their productions and are afterwards also composed into a complete solution. Clearly each transformation module (template, rule or attribute) produces a target language expression which is composed with the others to form a complete solution.

Merely using in-place substitutions to implement an invasive composition mechanism is very cumbersome and troublesome. Invasive composition mechanisms need to exert influence on various parts of other components in the system. Since only in-place substitutions are supported, developers are often tempted and forced to come up with creative work arounds for two problems (1) escaping from their local context to the other parts of the system and (2) implementing their effect in those parts of the system. When these two problems are dealt with naively the escaping and the implementation of their effect highly depends on the implementation details of the rest of the system and its components and thus on the state of the transformation process. Very soon these dependencies clutter up the system and result in a spaghetti code implementation. To keep this more or less manageable a staged transformation process is the most commonly used solution where the escaping and the implementation of their effect is performed in two separate stages. However, this does not reduce the number of dependencies.

Clearly transformation systems are not very suitable to implement invasive composition mechanisms. In order to remedy this situation, we believe that it is necessary to *extend* current transformation systems with a *suite of basic invasive capabilities*. These capabilities should not only facilitate the implementation but render it also more robust to evolutions of the other components of the software system.

Currently we are experimenting with a suite of basic invasive capabilities

based on the model presented by subject-oriented programming (SOP) [OKK+96, SCT99]. The extensions for the transformation systems we propose are thus based on SOP. SOP was formulated and founded in terms of object-oriented programming and introduced two kinds of rules: correspondence rules and combination rules. The correspondence rules declare which parts of the components must be combined with one another, the combination is performed by the combination rules. The two rules are externally defined to the components. To apply the SOP ideas in a general setting in transformation systems, a couple of modifications had to be made: (1) generalization and integration of those two rules into the transformation paradigm (2) additional context specifications expressed in declarative source language constructs using paths, (3) automation of tedious context specifications, (4) selection of the most specific and applicable rules and (5) increase of the robustness of the combination rules.

The above extensions are build on top of the Linglet Transformation System (formerly known as Keyword Based Programming [Cle03]). Since our incentive for this research lies in the need for more domain-specific concern-specific languages, the invasive composition mechanisms of current experiments are usually implicit constructs in those languages. Further experiments with other generative techniques are necessary to refine and validate our approach.

References

[Bat03] Steve Battle. Boxes: black, white, grey and glass box views of webservices. Technical Report HPL-2003-30, HP, 2003.

[BW97] M. Buchi and W. Weck. A plea for grey-box components, 1997.

[Cle03] Thomas Cleenewerck. Component-based dsl development. In *Proceedings of GPCE03 Conference, Lecture Notes in Computer Science 2830*, pages 245–264. Springer-Verlag, 2003.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428. ACM Press, 1993.

[OKK+96] Harold Ossher, Matthew Kaplan, Alexander Katz, William Harrison, and Vincent Kruskal. Specifying subject-oriented composition. *Theor. Pract. Object Syst.*, 2(3):179–202, 1996.

[OT00] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[SCT99] Harold Ossher Siobhán Clarke, William Harrison and Peri Tarr. Subject-oriented design: towards improved alignment of require-

ments, design, and code. *ACM SIGPLAN Notices*, 34(10):325–339, 1999.

[TG97] D. Tombros and A. Geppert. Managing heterogeneity in commercially available workflow management systems: A critical evaluation, 1997.

Metaprogram Implementation by Second Order Source Transformation

James R. Cordy*
Medha Shukla Sarkar†
School of Computing, Queen's University
Kingston, Ontario, Canada K7L 3N6
cordy@cs.queensu.ca, msarkar@mtsu.edu

Metaprogramming is the process of specifying generic software templates from which classes of software components, or parts thereof, can be automatically instantiated under direction of a formal design model to produce new software components. In the μ* system [Cordy92], metaprograms are specified using an annotated by-example style accessible to ordinary programmers of the target programming language. Annotations in the form of Prolog-like predicates specify the design conditions under which different parts of the source template are to be instantiated. Instantiation of a source component is then done by providing a design model as a database of Prolog facts from which the design conditions can be evaluated and source component instances automatically generated using Prolog-style deduction.

The implementation of μ* is interesting in the context of this workshop because it is entirely done using the source transformation language TXL [Cordy91,04]. The implementation is achieved in a two stage process in which metaprograms are first translated by source transformation into equivalent TXL programs. These TXL programs are then run with input from a design database to implement instantiation of the metaprograms and generate instantiated source code by source transformation. In essence, this implementation is a second order source transformation.

1. μ* : A Family of Metalanguages

μ* (pronounced "mew-star") is a family of by-example metaprogramming languages that share a common metanotation and implementation. The philosophy of the family is exactly the ideal: the metaprogramming language for each target language consists of the target language itself, augmented with meta-annotations specifying conditions in the design database. For example, μC, the metalanguage for C, consists of C program syntax, optionally annotated with meta-annotations. The syntax of meta-annotations is the same across all target languages. In each case, the syntax of the basic metalanguage is the syntax of the language itself, and the syntax of the meta-annotations is the syntax of μ*. The target language can be any programming or specification language with a formal syntax.

2. By-example Metaprogramming

In μ*, every program written in a target language is a metaprogram unconditionally generating itself. Thus every C program is automatically a μC program, and every Prolog program is a μProlog program. Syntactically contained program fragments (for example, declarations, statements, and so on) are also in general metaprograms for themselves.

The addition of meta-annotations to a metaprogram attaches the metaprogram to the design database and makes generation of the annotated parts conditional on the facts in the database. The range of affected code dependent on a design condition is denoted by enclosing it in backslashes, followed by the meta-annotation and a double backslash to mark its end, as shown in Figure 1. The backslash is the only symbol reserved by μ* it can be replaced with any other single symbol.

Because in many cases the intended role of the affected area in the target source is ambiguous, the role must be given explicitly following the bracketed area, as shown in Figure 1. The role is the name of the intended part of speech in the target language reference syntax (that is, the common name of the entity in the target language, for example *statement* or *declaration* in C) enclosed in square brackets [ ].

3. Generative Metaprograms

The μ* annotation language provides two basic operations: *when*, which includes a section of target source conditionally on the provability of a predicate on the database, and *each*, which generates one copy of the section of target source for every solution to a predicate in the design database (Figure 2). These two operations can be nested to give complex combinations of conditional generation.

The database is searched for solutions to each annotation predicate. When a solution is found, the metavariables in the predicate are bound to the terms found in the solution in the design database. The metavariables can then be instantiated in the target source generated for that solution. Repeated instances of a metavariable in a predicate specify unification in the usual Prolog way, so the predicate *function(F[id])* ***and*** *returns(F,int)* specifies only those entities that are functions in the design that return the type *int*. Figure 2 shows an example specifying a μC metaprogram to generate external C routine declarations for every function entity in a design database.

When programmers write code templates, they often use a pseudo-code style in which descriptive identifiers take the place

```
const char *strsignal(int n)
{
    static char
        buf[sizeof("Signal ")+1+INT_DIGITS];
\
    if (n>=0 && n<NSIG && sys_siglist[n]!= 0)
        return sys_siglist[n];
    sprintf(buf,"Signal %d",n);
\ [statement*]
  when listing
\\
    return buf;
}
```

Figure 1. Trivial Example μC Metaprogram.

The if and sprintf statements enclosed in backslashes are conditionally included in instances of the metaprogram only if "listing" is a design fact in the design database.

---

* Author's present address: ITC-IRST, Trento, Italy.
† Author's present address: Middle Tennessee State University, U.S.A.

```
\
  extern FType F();
\ [declaration*]
  each function(F[id])
   and returns(F,FType[type])
\\
```

Figure 2. Trivial Generative μC Metaprogram.

The interpretation is that a sequence of declarations is to be generated, one for each "function" entity in the design database. Unification on design facts finds the associated type automatically from the "returns" design fact.

of sections to be filled in later. μ* provides this same feature by allowing metavariable identifiers to take the place of any part of a target source fragment enclosed in backslashes, and by allowing later refinement of the role and source text of the metavariable, either as part of the solution to a predicate, or by using a *where* clause.

A *where* clause is a nested metaprogram that generates a target source fragment and binds it to a metavariable for use in other parts of the metaprogram, for example, the main source text. While in this position paper we do not have room for realistic examples, the nested combination of *when*, *each*, and *where* with Prolog-style predicate solution and unification on design databases gives μ* great power and flexibility while retaining the by-example nature of metaprogram templates. It has been used to specify and generate complex code artifacts in C, Prolog and Turing using design databases describing production software interfaces such as OpenGL.

4. Implementation of μ* Using TXL

μ* is implemented using the TXL source transformation language [Cordy91,04] by translating each μL metaprogram for a target language L to a corresponding TXL source transformation ruleset using TXL source transformation. The generated TXL ruleset is then combined with reference grammars for the target language L and Prolog to create a TXL program that implements the instantiation of the μL metaprogram from a design database of Prolog facts, as shown in Figure 3. The translation of μL metaprograms to corresponding TXL metaprograms is itself achieved using a source transformation specified and implemented in TXL. In essence, this implementation is simply a second order source transformation interpretation of the original μL metaprogram.

The purpose of this position paper is to introduce and explore the possibility of generalizing this technique to the wider implementation of metaprogramming systems using source transformation tools. While in this application the technique is driven by an entity-relationship design database in Prolog form, there is no fundamental reason why the design model could not be represented in any other design notation, including those based on UML [OMG03]. And while the particular source transformation system used here is TXL, there is no reason why the technique would not work with other tools.

References.

[Cordy92] J.R. Cordy and M. Shukla, "Practical Metaprogramming", *Proc. CASCON'92, IBM Centre for Advanced Studies Conference,* Toronto, November 1992, pp. 215-224.

[Cordy91] J.R. Cordy, C.D. Halpern and E. Promislow, "TXL: A Rapid Prototyping System for Programming Language Dialects", *Computer Lang*. 16,1 (January 1991), pp. 97-107.

[Cordy04] J.R. Cordy, "TXL - A Language for Programming Language Tools and Applications", *Proc. LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools and Applications,* Barcelona, Spain, April 2004, pp. 1-27.

[OMG03] Object Management Group, http:/www.omg.org, *Unified Modeling Language Specification v1.5*, March 2003.

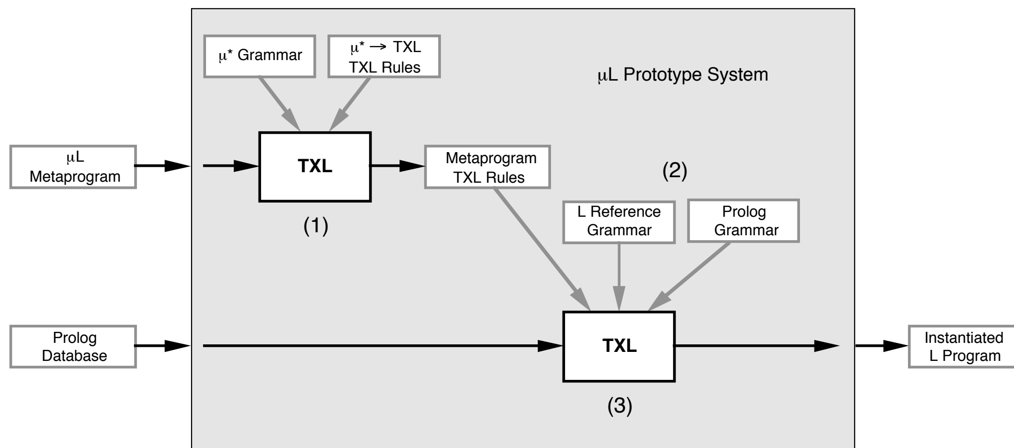


Figure 3. Implementation of μ* Using TXL.

A TXL source transformation is used to translate μL metaprograms to TXL transformation rules for a target language L (1). The result is combined with standard reference grammars for L and Prolog to give a complete TXL program (2), which is then run with a Prolog form entity-relationship design database as input. The design database is transformed by the TXL program to a target language L instantiation of the μL metaprogram (3). The entire process is very efficient, running in seconds on practical

Lexical Source-Code Transformation

Anthony Cox, Tony Abou-Assaleh, Wei Ai, and Vlado Keselj
Faculty of Computer Science, Dalhousie University
Halifax, Nova Scotia, Canada
{amcox,taa,weia,vlado}@cs.dal.ca

Abstract

As an alternative to syntactic matching on a program's abstract syntax tree, we explore the use of lexical matching on a program's source-code. Lexical techniques have been shown to be effective for the approximation of an abstract syntax tree, thus permitting tools that use regular expressions to effectively specify rewrite targets. In this paper, the features needed to support lexical rewriting are examined. As well, we report on several tools created to explore these features and suggest directions for future research.

1 Introduction

It is not always possible, or desirable, to parse a source file during maintenance. Errors, missing header files, or embedded language constructs can prevent parsing. Even if the file can be parsed, it might not be beneficial to do so, such as when performing maintenance on the macros in a C file.

Parsing is needed when a maintenance tool manipulates the source-code's abstract syntax tree (AST). As shown by Cordy *et al.* [5] AST transformation is an effective and versatile maintenance technique. The creators of the Software Refinery Toolkit [2] also support this view. However, tools such as TAWK [8], TXL [4], A* [10] , and Refine [2] use syntactic matching and must first parse a file to enable matching against the code's AST.

When code can not be parsed, an approximate AST can be extracted using a lexical technique [11, 6, 7]. This observation indicates that transformation-based maintenance is possible using lexical tools. We now examine the features that such tools should provide and then present tools we developed to explore these features.

2 Tool Features

To facilitate adoption, key tool elements should be well-known and highly accepted by programmers. Patterns and actions should be expressed using familiar formalisms. As well, tools should strive for simplicity; for example, by avoiding complex disambiguating rules. The following features adhere to these principles as much as possible.

Regular Expressions: Within the Unix community, patterns are traditionally expressed using regular expressions (RE). While other, specialized pattern languages exist, none are as widely used as RE. RE have been well studied and provide a strong candidate for a pattern language.

Unrestrictive Action Language: Actions should be specified in a complete programming language to avoid any limitations. Whether the actions are compiled, as in lex, or interpreted, as in AWK, is dependent upon the desired runtime efficiency of the tool. We currently advocate the use of a C-like language for its versatility and popularity.

Stream-Based Match: In source-code, the line structure (format) and syntactic structure are unrelated. Consequently, tools that use line-based, or record-based, matching are ineffective for code maintenance. Stream-based matching considers a file as a stream of characters, with newlines possessing no special properties. Cgrep [3] and grep (when used with the -z option) perform stream-based matching and can effectively locate syntactic constructs.

Disjoint Match: It is possible for two matches to overlap. For example, the expression `(ab)|(bc)` can be matched twice against the text `abc`. Disjoint matching resets the matcher after each match, thus giving priority to the first match in the stream. Anecdotal evidence suggests that this behaviour is what programmers most expect.

Unambiguous Matching: A RE is ambiguous if there exists a string that it matches in more than one way. For example, the expression `(abbc)|(ab*c)` has two matches against the string `abbc`. As it is possible to restructure every ambiguous RE into an unambiguous one [1], ambiguity can be removed, avoiding the need for complex disambiguating rules. In our experience, unambiguous RE are easy to write, once one learns where ambiguity occurs.

Shortest Match: Most RE matching tools use a longest-match algorithm, except for Perl [12], TLex [9], and cgrep [3], which support shortest-match. While both algorithms can match the same sets of strings, for extracting nested

```
dgrep -P -o -z '^[^\n]*?\n\w*main.*?^{.*?^}' file.c
```

Figure 1. Extracting a Function Definition with Dgrep

constructs, shortest-match is simpler (e.g. using shortest-match `{.*}` finds blocks in C).

Iterative Operation: When shortest-match is used, matching always finds the innermost of a nested construct. To locate all instances of the construct, iteration permits matching to occur from innermost to outermost. The iterative application of shortest-match simulates a bottom up traversal of the AST. A similar facility exists in A*, which has notation to specify multiple passes over an AST.

Sub-String Match: It is often useful to limit matching to a set of sub-strings identified in the text, such as matching RE_1 *contained-in* RE_2, or RE_1 *containing* RE_2. Cgrep uses *universes* to provide partial sub-string matching. Universes permit RE_2 (e.g. a variable) to be located in RE_1 (e.g. a block), supporting the 'contained-in' pattern. *Start states* in lex provide a similar functionality. Additional research is needed to explore the 'containing' pattern.

Other Features: Lex has many extensions to improve expressibility; `REJECT` permits matches to be rejected and `yyless` permits parts of a match to be returned to the text stream. TAWK, cgrep and lex, all use macros for pattern definition and simplification. LSME [11] permits sub-strings of a match to be named and accessed in actions. These features are also of value in a code rewriting tool.

3 Our Tools

Egret is a cross between cgrep and lex. Users create a lex-like specification that is processed and compiled to produce a transformation tool. Egret uses an unambiguous variant of cgrep's stream-based shortest-match algorithm. Cgrep's matching algorithm has been shown to be effective for the extraction of an approximate AST [6, 7]. Many of the features we have presented result from our experiences in implementing and testing Egret.

Dgrep is a modified version of GNU grep where the restriction preventing the use of the -z option with the -P option has been removed to demonstrate that stream-based match can be used with Perl's *ungreedy-match*. Perl differs from cgrep in that the leftmost criteria overrides the shortest-match criteria. For example, the pattern `a*b` when matched against `xaaabx`, returns the match `aaab` using Perl's ungreedy match and `b` using cgrep's shortest-match. Figure 1 demonstrates the use of dgrep to locate the definition of the function 'main' from code written using GNU stylistic conventions.

4 Future Work and Conclusion

The design and implementation of Egret identified many features needed by lexical source-code transformation tools. Dgrep explored code maintenance using Perl's RE syntax combined with stream-based matching. These implementations suggest that it would be useful to extend Perl to support unambiguous, stream-based, disjoint matching. Perl provides a rich set of programming facilities and offers many of the features needed for code transformation. Consequently, we believe the development of a Perl transformation module will provide an effective tool for performing lexical level source-code maintenance.

References

[1] R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, 20(2):149–153, 1971.

[2] S. Burson, G. Kotik, and L. Markosian. A program transformation approach to automating software re-engineering. In *International Computer Software and Applications Conference*, pages 314–322, Chicago, Illinois, October 1990.

[3] C. Clarke and G. Cormack. On the use of regular expressions for searching text. *ACM Transactions on Programming Languages and Systems*, 19(3):413–426, May 1997.

[4] J. Cordy, I. Carmichael, and R. Halliday. *The TXL Programming Language – Version 10.2*. TXL Software Research Inc, Kingston, ON, 2002.

[5] J. Cordy, T. Dean, A. Malton, and K. Schneider. Source transformation in software engineering using the TXL transformation system. *Journal of Information and Software Technology*, 44(13):827–837, 2002.

[6] A. Cox and C. Clarke. A comparative evaluation of techniques for syntactic level source code analysis. In *Asia-Pacific Software Engineering Conference*, pages 282–289, December 2000.

[7] A. Cox and C. Clarke. Syntactic approximation using iterative lexical analysis. In *International Workshop on Program Comprehension*, pages 282–289, June 2003.

[8] W. Griswold, D. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *International Workshop on Program Comprehension*, March 1996.

[9] S. Kearns. TLex. *Software Practice and Experience*, 21(8):805–821, August 1991.

[10] D. Ladd and J. C. Ramming. A*: A language for implementing language procesors. *IEEE Transactions on Software Engineering*, 21(11):894–901, November 1995.

[11] G. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, July 1996.

[12] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly, Sebastopol, CA, 3rd edition, 2000.

Software transformations supporting software engineering

Magne Haveraaen, Universitetet i Bergen (N) / University of Wales Swansea (UK)

Position paper for stsw04 at gpce04, Vancouver, Canada, Oktober 2004

1 Introduction

When writing software we are constantly faced with design decisions. These range from irrelevant syntactic variations to important choices on software organisation. Often we see a tension between what is *efficient* and what is *flexible and reusable*. So we end up phrasing the same information in many variations, causing a severe maintenance problem: how to keep all these variants synchronised when modifying them.

This can be seen as the "curse of syntax'. It is our belief that software transformation tools could ease it somewhat, allowing us to effortlessly move between alternative syntactic formulations, to combine code fragments for maximal reuse, and even to utilise stated information for a rich variant of purposes.

The rest of this position paper illustrates these ideas with a running example: sorting methods.

2 Notational variations

One of the fundamental choices we have to make when writing software is what programming language to use. This is strange, since most of what we will be writing is the same independent of this choice: in almost any imperative or object-oriented language, we will declare the same data structures and write down the same algorithms and method calls, only with slight variation in notation. Thus we should readily have tools which translated these code fragments from one language to another.

Even within a programming language we are faced with a meaningless set of choice. Assume I am implementing a method for sorting data in an object-oriented language such as C++. Then I will need to decide whether to write the method

- as a *function*, i.e., taking the unsorted data `b` as an argument and returning the sorted data `a` (in imperative or object-oriented form):
  ```
  a := sort(b);
  a := b.sort();
  ```
- as a *procedure* with an input parameter `b` of unsorted data and an output parameter `a` of sorted data (4 variants):
  ```
  sort(b,a);
  sort(a,b);
  b.sort(a);
  a.sort(b);
  ```
- as a *procedure with in situ* sorting of the data `a` (2 variants):
  ```
  sort(a);
  a.sort();
  ```
- as a *main program* which may receive its data from file parameters (or input and output stream), by prompting the user, etc.

All of these variants have their merits, depending on the use we want to make (a sorting function in an expression, a sorting statement in a loop, sorting as part of a shell, etc.), or based on what kinds of data we will be manipulating (e.g., arrays for in situ sorting). Such variants can be generated from a single source, thus making our implemented algorithms more reusable. Further, we should be able to transform between expressions using a function calls and statements using procedure calls, ensuring that the two forms compute the same results, giving us full freedom to express our code without regards to what may be more efficient in a given context.

In the above we have ignored variations of parameter passing mechanisms: by value, by reference, by name, by indirection (pointers), ... It should be obvious that such variants can be generated by a suitable tool, given that our intent with the parameters are clearly stated (data coming in, data going out, or data to be updated in situ).

3 Software organisation

The internal organisation of software is important for maintainability and reusability. Over the last few decades (since the early 1970s), it has become clear that using abstract data types ($\approx$ classes – a data structure declaration together with all the methods that access its components directly) is favourable in most situations. So we will provide classes like

```
template<class T> Array { ...
   quicksort(); selectionsort(); ...}
   // implementation based on accessing
   // elements by computing address
   // references

template<class T> List { ... quicksort();
   selectionsort(); insertionsort(); ...}
   // implementation based on unhooking
   // and rehooking list pointers
```

The intimate relationship between the data structures and the way we implement the algorithm makes this natural. But what if we want to change the pivoting strategy of the quicksort algorithms. Then we would like to see all the quicksort routines grouped for editing, which would be easy with an integrated transformation and editing tool. Or, more ideally, implement the pivoting strategies as separate code fragments which are then (invasively) composed with the quicksort methods.

We would also need projections giving us a view with all sorting methods, e.g., for adding an extra parameter indicating whether we want ascending or descending ordering of the sorted result. As part of this modification we must also augment all existing calls to the sorting methods with an extra parameter with the *ascending* value – a typical transformation job.

4 Optimisation

Many of the issues discussed above are really optimisation issue, e.g., in situ modification versus copying of data, choice of parameter passing mechanisms, choice of which pivoting algorithm to use. From a software engineering viewpoint such choices should not be forced upon the code developer, but be handled at a configuration / optimisation stage, i.e., by a software transformation tool.

Traditionally optimisation has been the realm of compilers which would have an intimate knowledge of the target hardware. But often compilers are put off from doing a good job by trivial obstacles due to the semantic complexity of the source programming language. A transformation tool may be used to remove these obstacles, taking into account specific knowledge about programming styles and software domains in a project. Thus the tool may, e.g., unfold method calls inside loops, rewrite expressions on large data structures with equivalent imperative code, replace a sequence of method calls by a more efficient sequence, even taking into account properties of specific data sets.

Such transformation rules will in part be generic, but many of them will be very domain specific. The latter requires easy extension of the transformation tool, e.g., by making it sensitive to formal specifications (user defined rules) given with the code. Then strategies may be provided which take into account characteristics of the target computer/compiler, allowing the transformation tool to choose the more optimal code to feed the compiler.

5 Conclusion

We have sketched several areas where software transformation systems may aid the software engineering aspect of software development and maintenance. Many of them relate to the semantic information expressible in formal specifications.

Some of these ideas have been pursued in the software transformation system CodeBoost (see `http://www.codeboost.org`), others will be investigated in the future.

Generic Software Transformations

Jan Heering[1] and Ralf Lämmel[2]
[1] CWI, Amsterdam, Jan.Heering@cwi.nl
[2] CWI and Vrije Universiteit, Amsterdam, ralf@cwi.nl

A generic software transformation is a *transformation scheme* that can be instantiated to an actual transformation by supplying a language and a language concept as arguments. For instance, a generic **extract** might be instantiated to

- **extract**[C, variable] (common subexpression elimination in C);
- **extract**[EBNF, non-terminal] (elimination of common parts of right-hand sides of EBNF syntax rules);
- **extract**[C, function] (function folding in C);
- **extract**[Java, method] (method folding in Java);
- **extract**[C++, template] (folding class definitions into template instances in C++).

Generic transformations capture the common principles underlying transformations across different languages as well as different constructs for the same language. The languages involved are not limited to general purpose programming languages, but also include domain-specific languages (like EBNF) and modeling languages (like UML).

Generic transformation primitives Some categories of language constructs, such as abstractions and applications, immediately suggest primitives for building generic transformations. In fact, the extraction illustrated above can be broken down in a sequence of primitive transformations for *introduction* and *folding* [2]. Introduction amounts to insertion of a new abstraction (for instance, a method) whose body is equivalent to the fragment of interest without creating any conflicts. Folding replaces the fragment of interest by the application of the abstraction (for instance, a method call). Their inverses, elimination and unfolding, are themselves again primitives.

Crosscutting language concepts like scope and typing suggest further primitives. A simple one for typing is adding a type annotation for languages with type inference. That is, an inferred type is made explicit in the source code. Scoping primitives affect the scope of bindings or the way entities are brought into a scope. A simple example is moving a local declaration to the top level so that it becomes global. Another example is replacement of a hard-wired reference to a declaration by a reference to a newly introduced parameter.

Other primitives concern more specific constructs or concepts that are nevertheless part of many languages. These capture conditional algebraic laws for the introduction and elimination of the constructs in question. For instance, subtype polymorphism is found in many object-oriented languages. There is a correspondence between cascaded conditionals and polymorphic dispatch, which leads to transformations for their

elimination and introduction. Another example concerns regular expression operators, which are found in many languages. Again, the elimination and introduction of regular operators can be described in a generic way, while EBNF to BNF and BNF to EBNF conversions are specific instantiations of this idea.

Generic code smells The notion of *code smell* was introduced in the context of object-oriented software refactoring, but a generic notion that is parameterized with a language and a *criterion* or *metric* is applicable to the much wider class of languages outlined above as well as to other types of transformations. For object-oriented languages and static or structural criteria, (a special case of) the original notion is obtained. Other criteria may be related to security or performance. Such smells indicate parts of the code that may benefit from security enhancing or optimizing transformations. Some possible criteria and corresponding smells are:

- distance, e.g., too many levels of blocks between declaration and use,
- coupling (in the sense of object-oriented programming),
- size, e.g., methods that are too long (in terms of lines of code),
- style, e.g., missed opportunities for using higher-level idioms,
- extensibility, e.g., fragile use of hardwired conditionals,
- generality, e.g., fixed variation points including overly specific types,
- readability, e.g., too concise use of higher-order functions.

Generic transformation frameworks Current work on semantics is yielding techniques for the modular definition of language concepts. A generic transformation framework is supposed to support a similar degree of modularity. In fact, the underlying ontology of such a transformation framework has to be aligned to the ontology of language concepts. More than this, we seek strictly aligned definitions of semantics and transformations. That is, a complete framework for language concepts must provide all of the following: syntax, static analysis, execution semantics (if any), primitive transformations, and possibly other ingredients related to the use of transformations in actual scenarios. In [1] it is argued that *equational logic* can serve to unify the definition of static analysis (as abstract interpretation), execution semantics, and program transformation. The benefits of such a unified view are considerable. In particular, equational specifications of interpreters, transformation systems, or analyzers can often be done in a modular fashion by adding or removing sets of equations.

References

[1] J. Field, J. Heering, and T. B. Dinesh. Equations as a uniform framework for partial evaluation and abstract interpretation. *ACM Computing Surveys*, 30(3es), September 1998. Electronic supplement: 1998 Symposium on Partial Evaluation.

[2] R. Lämmel. Towards generic refactoring. In *Proc. 2002 ACM SIGPLAN Workshop on Rule-Based Programming (RULE '02)*, pages 15–28. ACM Press, 2002.

Towards comparing transformation systems and formalisms

Torbjörn Ekman Görel Hedin

18th June 2004

1 Introduction

There is a need for comparing and evaluating software transformation systems. Potential users would like to compare different systems in order to select a particular one for a particular use. Tool builders would like to easily learn about other systems and to compare with their own in order to exchange knowledge and ideas and to improve their tools.

We propose that there should be a collective effort among tool builders in trying to formulate and implement a number of software transformation benchmark examples. Each benchmark could be specified as a set of textual input-output test cases, together with some informal explanation of the language and transformation. To allow the easiest comparison, a solution for a particular tool could be given as a set of text files that specify the transformation, together with a short pdf description giving an overview of the modules and tools involved in the solution. The text-based input-output will allow many different kinds of tools to be compared, including hand-written solutions.

Questions that pop up in this context are: what benchmarks should be chosen? What properties are interesting to compare? Can we find some useful categorisation of the examples? What should be the work process of this collective effort? In this position paper we present our own background and give some input to the discussion of these questions.

2 Our background

We have a background in compiler construction based on attribute grammars. We have developed ReRAGs, a declarative conditional rewrite formalism that is based on object-oriented programming, static aspect-oriented programming, reference attributed grammars, and conditional in-place rewriting. The technique has been implemented and tried out on substantial applications, including the development of a Java 1.4 compiler. The main advantages of our approach are the following

- The possibility to use context-sensitive conditions for transformations. The context-sensitive conditions are computed by reference attributed grammars, and thus have access to the complete AST.
- Declarative rewrites that are not explicitly scheduled, but are run automatically as needed.
- An evaluation method that can handle interleaved rewrites and context-sensitive computations
- Reasonable performance. Our generated Java compiler is around 4 times slower than javac.
- The specification language is a DSL on top of Java.
- High support for modularity and extensibility.

3 What benchmarks should be chosen?

It is important that the benchmarks are formulated in such a way that they are easy to understand by potential users, and that they are fairly tool neutral and sufficiently small so that several tool builders will take on the extra work of implementing them.

An example benchmark could be a small rename refactoring. The input could be a specification of what declaration to rename, followed by the original program. The

output could be the refactored program or a message "precondition violated" if there is already a declaration of that name.

```
Testcase 1
Input:
rename (a to c) in
program
   void a() { int b=0 }
   void b() { int a=a() }
end
Output:
refactored program
   void c() { int b=0 }
   void b() { int a=c() }
end

Testcase 2
Input:
rename (a.b to c) in
program
   void a() { int b; b=0 }
   void b() { int a }
end
Output:
refactored program
   void a() { int c; c=0 }
   void b() { int a }
end

Testcase 3
Input:
rename (a to b) in
program
   void a() { int b }
   void b() { int a }
end
Output:
precondition violated
```

The benchmark above may be easy to implement in some tools and very difficult in others. Some tool builders might like to change the input/output specification for some reason. Some other benchmarks could be a tiny aspect weaver, template instantiation, a simple DSL. One might also consider more traditional compiler-oriented tasks like name analysis, type checking, code generation.

4 What properties are interesting to compare?

There are many properties that can be interesting to compare between the different solutions to a given benchmark. For example,

- Specification size
- Transformation performance
- Modularity and extensibility of the specification. Are the modules composable?
- What characterizes the specification? (pass-oriented, imperative, declarative, etc)

5 Benchmark categorization

It might be useful to categorize the benchmarks in order to more easily understand the space of problems that they span. What dimensions of categorizations could be useful? One possibility could be the cardinality of the transformations, e.g., are single fragments transformed to new single fragments (one-to-one), single fragment to several new fragments (one-to-many), and so on. Can the benchmarks be characterized as context-dependent vs. context-free? For example, the renaming benchmark might be categorized as one-to-one and context-dependent. What other dimensions might be used for categorizing the benchmarks?

6 Collective work process

We do not expect the answers to our questions to be found in a single workshop session. A continuous discussion over a series of workshops at various conferences will be needed. Between meetings, there can be individual activities coordinated via some web forum such as a wiki. It is important that the process is open and new benchmarks and ideas can be added and evolved over time.

Tracing Abstractions through Generation

Karl Trygve Kalleberg
Department of Informatics
University of Bergen
karltk@ii.uib.no

2004-06-29

1 Background

When combining abstraction mechanisms native to a programming language, one can usually rely on complete and sensible debugging support. A regular debugger for an imperative language knows about functions, records, unions, and so on. It is even the case that common mistakes are explained at compile-time. Examples of this is the Jikes compiler[1] for Java and the MIPSPro compiler for C++, which both emit detailed semantical errors and warnings.

As generative programming is frequently used to add new abstraction features to existing languages or generate large parts of the code from formal specifications, the language compiler's output often becomes less relevant: If the warning pertains to code inside a programmer-editable template, the programmer may fix the problem, but often the compiler warns about issues inside automatically generated code which the programmer has no way of changing, unless he is also a developer of the generative programming system itself.

2 Some common problems

The problem is frequently compounded by the following factors:

- *Code as text*: The generative system treats code as text, and operates entirely on strings.

  The appeal of this approach lies in its availibility (there are literally hundreds of text preprocessors) and general nature (not tied to any particular language).

  The string operations do not treat the text as structured, which frequently results in syntactical errors, such as a missing '}' to close a code block.

  Most pre-processor systems fall into this class, with Java ServerPages[2], PHP[3] and the C/C++ preprocessor being well-known examples.

- *Late semantical analysis*: The generative systems may treat the code as synactic trees, usually operating on a CST or an AST representation of the

underlying language. It is rare however, that the rewrite rules have access to full semantic information for the elements it rewrites, and static semantic analysis is often left as an excercise to the compiler. Notable exceptions are the Eclipse JDT 3.0 for Java[4] and CodeBoost for C++ [5],[6], which offer (close to) full semantic analysis to the rewrite rules.

- *Step-wise rewriting*: Generative techniques may often generate code that, when sent to the compiler, is very different from the individual components it was generated from, making traditional debugging difficult.
- *Dynamic generation and loading*: In some forms of generative programming, the assembled binary code is produced on-demand inside a running system, presenting opportunities for undiscovered bugs, even syntactic ones, after customer deployment time. These flaws are supposed to be caught by various coverage analyses, but experience shows that this is difficult in practice.

The problem of abstraction traceability has been known since the inception of compilers, and is also a big issue in world of modelling these days. For both imperative and object-oriented languages, annotating the produced assembly code with line number (and source code excerpts) proved effective, and has become a de facto part of most compilers.

The annotation solution cannot easily be applied to generative programming in the general case, however, as the coupling between component (or template) source code and the final, generated product is extremely loose. What appears as one line in one component may end up at hundreds of places after generation.

This problem bites most forms of code generation and compositions, in particular implementors of domain-specific languages (DSLs) as well as aspect-oriented programming.

3 Some suggested solutions

In addition to experience from embedded DSLs in languages with extensive metaprogramming capabilities, such as Lisp, we have:

- *Annotations coupled with tool support* If annotations are available for inspection by tools inside the final product, inspectors can be written that relate generated code to initial components and specifications. This has some relation to debug slicing [7].
- *Rigorous synactical and semantical analysis at rewriting time* Provided with full-fledged semantic analysis, and possibly domain-specific rules for the particular components which encode their semantics, errors would be caught at earlier stages, where the distance between the initial components and the final source code is smaller.
- *Step-wise rewriting with inspection* Interactive playback of the rewrites may also be helpful to track the origins of particular problem areas.

References

[1] Jikes Java Compiler . http://www.research.ibm.com/jikes/, 2004.

[2] Java ServerPages. http://java.sun.com/products/jsp/, 2004.

[3] PHP. http://www.php.net/, 2004.

[4] Eclipse Team. eclipse.org, 2004.

[5] Otto Skrove Bagge. CodeBoost: A Framework for Transforming C++ Programs. Master's thesis, University of Bergen, Norway, 2003.

[6] Karl Trygve Kalleberg. User-configurable, high-level transformations with CodeBoost. Master's thesis, University of Bergen, Norway, 2003.

[7] István Forgács Tibor Gyimóthy, ÁrpÁd Beszédes. An efficient relevant slicing method for debugging. In M. Lemoine O. Nierstrasz, editor, *Rewriting Techniques and Applications (RTA'02)*, Lecture Notes in Computer Science, Toulouse, France, September 1999. Springer-Verlag.

The Role Of Template Engines in Source Translation Systems

Terence Parr
University of San Francisco
parrt@cs.usfca.edu

The documentation for systems that generate or translate programs often use exemplars to describe their output format. These exemplars should become part of the translator itself as formal "templates" to describe the set of system output sentences. Just as we use grammars to describe input languages, templates should be used to describe output languages.

Many commercial translation systems use ad hoc output generators that amount to large pieces of code interwined with print statements. Retargeting and maintaining these beasts is truly a nightmare. In contrast, there are many fine translation engines that structure output generation as a series of formal rewrite rules. Many programmers, however, find these systems unpalatable precisely because of the sophisticated rules--the average programmer is uncomfortable with tools that have "black box" inference engines or similar powerful mechanisms.

The goal of my research is provide language translation tools that balance sophistication with a simplicity amenable to the needs of the average programmer. This short paper briefly describes a tool called StringTemplate (http://www.stringtemplate.org) and its accompanying philosophy of strict separation between output sentence specification and translation logic (in object-oriented terms, one would say "strict model-view separation").

BENEFITS OF STRICT SEPARATION

Compilers neatly separate optimization and analysis from instruction selection usually via a bottom-up instruction tree rewriting system such as TWIG or BURG that accept tree grammars. Retargeting the compiler is a matter of providing another tree grammar. Analogously, programs that generate text-based high-level code should isolate the translation logic from the mechanism for generating output sentences.

While there are numerous template engines for every popular language, almost none enforce this model-view separation. Strict separation guarantees that the output format is completely encapsulated in the template files and, equally importantly, that the translation logic is completely encapsulated in the code generator. The benefits:

- easy to understand, build, modify; you do not have to imagine the emergent behavior--you have an exemplar.

- easy to retarget; just swap in new template file as templates do not contain any logic.

- single point of change for optimizations and other logic.

- easy to maintain; changing templates is safer than changing code.

- better division of labor; e.g., I am building ANTLR 3.0 CodeGenerator.java and collaborators are building the various target language template files. Further, individual users can tweak the output to optimize ANTLR output for their situation.

- template reuse; templates without logic are merely presentations and can easily be reused by programs with similar logic.

I have shown elsewhere that such a ruthless restriction (no logic in

the template) does not emasculate the power of a template engine; a system such as StringTemplate comfortably generates languages beyond the context-free class into the context-sensitive.

In my experience using template-based code generators (for both large web servers and programming language generators), the strict separation of model and view is essential for the scalability and maintainability of the application.

Beyond this philosophical or strategic characteristic, there are numerous tactical advantages. For example, separating order of presentation from order of computation is a huge advantage. Templates let the logic code (controller) compute things when it is convenient or efficient rather than in the order imposed by the output phrase structure. The controller walks the structures in the model, computing the raw attributes in the most convenient manner. For example, when generating a C program, all functions must be declared before they are used. It is often the case, however, that you do not know the complete set of functions before you begin generating them. With a template you can set the attributes in any order; that is, you can add to the declarations list as you generate functions. The actual output is not generated until you ask the template to render to string.

StringTemplate

A StringTemplate is a "document" with holes in it where you can stick values. StringTemplate breaks up your template into chunks of text and attribute expressions enclosed in angle-brackets: <attribute-expr>. Expressions may reference attributes, may conditionally include subtemplates, may invoke other templates recursively, and may apply templates to lists of attributes. The language is distinctly functional in nature. StringTemplate.toString() evaluates all expressions, recursively walking nested templates to render to string. Here is a simple example using StringTemplate directly in code:

```
StringTemplate query =
  new StringTemplate("SELECT <column> FROM <table>;");
query.setAttribute("column", "name");
query.setAttribute("table", "User");
String output = query.toString();
```

More commonly for source translators, you will use a StringTemplateGroup which is just a list of templates defined in a group file. The ANTLR 3.0 prototype generator has a single CodeGenerator engine and multiple template group files, one for each target language. For example, here is the overall structure of a parser as implemented in Java:

```
parser(name, tokens, rules, DFAs) ::= <<
class <name> extends Parser {
    <tokens:{public static final int <attr.name>=<attr.type>;}>
    public <name>(TokenStream input) {
        super(input);
    }
    <rules; separator="\n">
    <DFAs>
}
>>
```

where

```
<tokens:{public static final int <attr.name>=<attr.type>;}>
```

is applying an anonymous template "{public static ...}" to each value

in a potentially multi-valued attribute called tokens, which was computed and sent in by the CodeGenerator. There is not a single string literal in the CodeGenerator that gets emitted to the generated file; output phrases are completely contained within the template group file.

SUMMARY

I have found StringTemplate to be an effective code generation aid for both large web sites and source translators such as the new ANTLR 3.0 prototype. The principle of strict model-view separation, StringTemplate's distinguishing characteristic, provides numerous advantages (in areas such as clarity, retargeting, and maintenance), all without crippling the power of the engine.

Transformation Circuits: Exploring new paradigms for Software Transformation Systems

Marcelo Sant'Anna
Advus Corporation
santanna@advus.com

1 AUTHOR'S BACKGROUND

The author has been working and developing software transformation systems during the last 15 years. He has been the main architect of some systems like Lavoisier, Draco-PUC and SpinOff. These systems had been used for building different kinds of software transformers, ranging from little language compilers to multi-language reverse engineering tools. He also worked with transformation systems from other authors like Draco-UCI, DMS, TXL, Refine and Popart.

2 POSITION

Developing himself a large set of transformers and providing support to developers that use the tools he built, the author got a special interest on how to tame complexity for designing and maintaining software transformers. Although regular Software Transformation Systems provide flexibility and functionality that ease the tasks of building transformers, the author thinks that the field is still on its infancy in terms of productivity, standards and common shared knowledge among researchers and practitioners.

During the workshop the author would like to discuss why the field still did not give a more expressive contribution to software engineering. Understanding that automation is essential for software engineering to mature, it is really intriguing that transformation systems did not make an impact.

Generally speaking, software generators and automation tools are in common use these days, but they are mainly handcrafted using standard programming languages. Sometimes the efforts to build such tools are so daunting that projects are dropped altogether. Software transformation systems could come to rescue, mainly on industrial projects where a strong time-to-market pressure is present and where companies need support to their product lines.

If we take into account the efforts revolving around Model-Driven Architectures (MDA), it is quite clear that most automation efforts are still very restrict. From a transformation systems standpoint, it is as if these industry efforts were completely missing what has been done in the field of transformation systems for the last 20 years.

3 WHAT ARE WE MISSING?

Given the social aspects, it seems we lack a good sense of community. Share of knowledge in the field had been quite pale. The topic is almost not explicitly discussed on major conferences and the good results on the field are definitely not well promoted.

Given the technical aspects, we all should ask ourselves if we are really delivering the promise of software automation. If we are not, what could be done? Why is still so difficult to educate a software engineer on the techniques we use. Why is the learning curve still so steep?

The author also understands that although we have a lot of great tools, we still lack good abstraction mechanisms to tame the intrinsic complexity of transformers. A more frequently exchange of experiences as well as discussion forums would help ideas to converge.

No matter how good parser generators, efficient tern rewriting engines and box-formatting pretty-printers we have, we should go beyond the common architectures and approaches and explore new options.

The author in particular is exploring the notion of Transformation Circuits. The idea is to inject ideas from the Signal Processing community into transformation systems. Taking a look at VLSI advancements it is quite clear that electronic engineers have been developing good models to tame the complexity of circuit integration. Given the fact that large part of IC engineering involves some sort of software programming, the author is interested on studying how electronic engineering techniques could help our field. A prototype called SpinOff had been developed and it has been used as a workbench for new ideas. The TC approach looks at transformers as if they were filters on a stream of data. The streams of data, representing programs to be transformed, flow on a set of circuits that operate on top of a shared bus. Functionality can be extended by plugging new filters into the bus and wiring them together.

At the workshop the author would like to discuss the approach, share his knowledge, and provide a demo on the concepts that he is exploring with his prototype transformation system.

For a reference on the author's view of the field shared with other professionals, please, take a look at http://www.dur.ac.uk/CSM/STS/

Understanding Model Transformation by Classification and Formalization

Shane Sendall, Rainer Hauser, Jana Koehler, Jochen Küster, Michael Wahler

IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
{sse, rfh, koe, jku, wah}@zurich.ibm.com

Software modeling techniques offer a means to address the size and complexity of modern day software problems through the use of abstraction, projection, and decomposition. Typically, multiple models are used to describe non-trivial software systems. However, if such models must be related and kept consistent by hand, then the viability of modeling as a means to reduce risks, minimize costs, improve time-to-market, and enhance product quality is nullified (most probably made even worse). As such, model-driven development approaches, in the direction of OMG's Model Driven Architecture initiative [OMG03], must be supported by tools that are at least able to automate the various tasks of keeping models consistent. Furthermore, the more that the various activities of model elaboration, synthesis, and evolution can be automated the more the above stated factors will be better addressed.

Our team at IBM ZRL [BPIA], entitled Business Process Integration and Automation (BPIA), is involved in research and development of model transformation techniques for model-driven development approaches in the domain of ebusiness solutions [GGK+03, HK04, KHK+03]. We are performing work on transforming business-level models to IT-level models and we are involved in the QVT-Merge submission for OMG's Meta Object Facility (MOF) 2.0 Query/View/Transformation Request for Proposal [OMG02]. Our effort in the latter area consists of work on a standard model transformation language for transforming MOF models, where MOF is an OMG standard for defining meta-models, i.e., the abstract syntax of a modeling language.

The problem of model transformation is similar to the one of program transformation and it also makes use of metaprogramming techniques [CH03, SK03]. However, it takes a slightly different direction by working with object-oriented metamodels, which define object graphs rather than syntax trees. We define the term model transformation in the following way: *A model transformation is a mapping of a set of models onto another set of models or onto themselves, where a mapping defines correspondences between elements in the source and target models*.

There are a number of different contexts of use that are applicable to QVT model transformations [Omg02]; these can be broken into two broad categories, inspired by Visser's classification for program transformation [Vis01]: *language translation*, and *language rephrasing*. In the former, a model is transformed into a model of a different language, i.e., a different model, and in the latter, a model is changed in some way, which may involve producing a new target model with the changes (distinct models) or changing the existing source model (single working model).

Like in [Vis01], language translation can be further sub-divided into *migration*: a model is transformed to another one at the same level of abstraction; *synthesis*: a model is transformed to another one at a lower level of abstraction; and *reverse engineering*: a model is transformed to another language at a higher level of abstraction.

Language rephrasing can be sub-divided into *normalization*: a model is transformed by reducing it to a sublanguage; *refactoring*: a model is restructured, improving the design, so that it becomes easier to understand and maintain while still preserving its externally observable behavior; *correction*: a model is changed in order to fix an error; and *adaptation*: a model is changed in order to bring it up to date with new or modified requirements.

The mapping between models established by the transformation may be required to be preserved over time. We call this characteristic of transformation *synchronization*

[GGK+03]. Examples of synchronization include: round-trip engineering and views. As part of synchronization, propagation of changes to a model may be made in one or more directions. Synchronization may be activated in a strict or loose fashion. *Strict synchronization* requires all changes to models to be taken into account immediately or in the next consistent state, e.g., views. *Loose synchronization* makes no statement on when synchronization should occur.

There are many different approaches available for model transformation; some of these include: relational/logic, functional, graph rewriting, generator/template-based, and imperative [CH03].

Our premise is that the different categories of model transformation in the QVT space are suited to different languages and approaches. As such, we believe that we should move towards understanding the requirements of each category and look at which kind of language is suited to which subset of problems. In doing so, we would like to understand the common requirements and also those that differ, and eventually build languages that are specifically address those specific problems.

Some questions that we are interested in addressing/discussing include:

- The field of compilation has a well understood categorization of languages. Building upon this work, how can one effectively formalize the different usage categories in the QVT space and the different model transformation languages so that we can more rigorously understand which ones match which domain? How "declarative" can we make a language targeted for such domains? What further categorization could we do with such formalizations?
- Bi-directional synchronization is a difficult problem in general. What existing approaches offer solutions? Is bi-directional transformation equivalent to uni-directional transformations in either direction? How do you avoid clobbering existing information on the return trip? How can trace information help in the return trip?

References

[BPIA] Business Process Integration and Automation, IBM Zurich Research Labs, Switzerland, 2004. http://www.zurich.ibm.com/csc/ebizz/bpia.html

[CH03] K. Czarnecki, S. Helsen; "Classification of Model Transformation Approaches". Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, USA, 2003.

[GGK+03] T. Gardner, C. Griffin, J. Koehler, R. Hauser; "A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard". MetaModelling for MDA Workshop, England, 2003.

[HK04] R. Hauser, J. Koehler; "Compiling Process Graphs into Executable Code". GPCE-04, 2004.

[KHK+03] J. Koehler, R. Hauser, S. Kapoor, F. Wu, S. Kumaran; "A Model-Driven Transformation Method". EDOC 2003, pages 186-197.

[OMG02] Object Management Group; "Request for Proposal: MOF 2.0 Query / Views / Transformations", 2002. http://www.omg.org/docs/ad/02-04-10.pdf

[OMG03] Object Management Group; "MDA Guide Version 1.0.1". 2003.

[SK03] S. Sendall and W. Kozaczynski; "Model Transformation - the Heart and Soul of Model-Driven Software Development". IEEE Software, vol. 20, no. 5, September/October 2003, pp. 42-45,

[Vis01] E. Visser; "A Survey of Strategies in Program Transformation Systems". Electronic Notes in Theoretical Computer Science, eds. Gramlich and Lucas, vol. 57, Elsevier, 2001.

Extending languages with transformation and generative technology

Ganesh Sittampalam
Oxford University Computing Laboratory
ganesh@comlab.ox.ac.uk

November 7, 2004

1 Introduction

There are great potential benefits for software engineering if we can make it easy to extend existing programming languages with new features, and provide efficient implementations of those features. The compiler optimisations required for this are best expressed at a variety of different levels; for example, some may be best viewed as transformations of the source code, others on some intermediate representation such as SSA, and others as peephole optimisations of the object code. This suggests that we should view the compilation process as a series of generative steps, in which one level of representation is compiled to a slightly lower level, interleaved with optimisation steps which transform code in a particular representation to an improved version in the same representation.

2 The AspectBench Compiler

Recently, I have been involved with the development of the AspectBench Compiler ("Abc") [1], an extensible compiler for the AspectJ [3] language, building on two existing frameworks, Polyglot [4] and Soot [5].

AspectJ is an extension for Java for introducing "cross-cutting" concerns. From the perspective of a language implementor, its important features are that it allows the programmer to write "aspects" which contain code which is added into existing Java source or bytecode, both in the form of completely new class members and of code added to existing methods. Conceptually, this happens at runtime, but in practice an efficient implementation requires compile-time "weaving" (or perhaps a modified JVM, but this route has not been explored much so far).

Polyglot is used for building frontends for extensions of the Java language, which are semantically checked and translated into Java before being handed to a standard Java compiler. Implementation of language extensions by translation to the host language is sometimes inadequate, and this is one such case;

the AspectJ language requires support for modifying existing programs only available as Java bytecode.

Soot is an analysis toolkit for Java which can read and write Java bytecode and convert it to and from a number of (Soot-specific) intermediate representations. One of these representations is *Jimple*, a simple three-address code with a close connection to Java bytecode (but none of the complexity inherent in having an implicit stack). Soot has recently been extended to provide a back-end named Java2Jimple for Polyglot that, naturally, transforms the Java output into Jimple, thus allowing Polyglot to be used as an end-to-end Java compiler.

The benefit of this for our implementation of AspectJ is that we can do much of the "weaving" at the Jimple level; the Jimple can be generated either from Java source or from bytecode, but the same weaver can be used on either. Of course we could also choose to weave directly into bytecode, but this is much more complicated and error-prone. Crucially, Jimple is also a language that is well suited to optimisation, and indeed a number of standard Java optimisations (both intraprocedural and interprocedural) have already been implemented in Soot; we have also implemented a variety of optimisations that take advantage of specific properties of AspectJ-produced Jimple.

Thus, the architecture of Abc is as follows. AspectJ code is read in from source and is transformed into Java, as well as some extra "aspect information", using a Polyglot extension. This Java is the converted into Jimple using Java2Jimple. Java code is read in either as source or as bytecode and transformed to Jimple, either via Polyglot and Java2Jimple or via the Soot class file reader. Next, the "aspect information" is used to weave the extra aspect-introduced code into the relevant Jimple, and a number of optimisations are run. Finally the Jimple is converted to bytecode and output as class files.

3 Evaluation

This overall approach has been very effective, in my opinion; it has led our team to the point of releasing a full AspectJ compiler within just eight months of starting work on it. Moreover, our compiler leverages the existing features of both Polyglot and Soot, making it relatively easy to implement both new language extensions on top of AspectJ, and to develop more sophisticated optimisations for the AspectJ language (there are various features of AspectJ that impose quite a significant runtime overhead [2], so such optimisations are worthwhile).

Some limitations have also become apparent, and I believe that addressing these would make it even easier to undertake a similar project in future.

- Polyglot is based on multiple tree rewriting passes, with extensible visitors and nodes to allow language extensions to be supported and new transformation passes to be added. This approach fits relatively well with Java, but is somewhat inefficient and leads to unnecessary work on the part of the language implementer in scheduling the passes. I believe an attribute-grammar based system would be much more effective.

- As many before us have realised, generating fragments of intermediate code (in our case Jimple) by manually building up the abstract syntax tree is verbose and rather error prone. Better quoting and static typing support for the generative code would be of enormous benefit.
- In addition, we have often found that we accidentally produce invalid Jimple code which then leads to invalid class files, and the error is only detected when we try to run the generated programs. More aggressive validation of intermediate results is very important for debugging, and whilst we did belatedly develop a validator for Jimple, much time could have been saved by doing so earlier. It is interesting to note that the need for this only become apparent when Jimple started to be used generatively; it has been used in Soot for analysis and transformation for many years without this becoming an imperative.

References

[1] The AspectBench Compiler, 2004. Will be availble from URL: `http://abc.comlab.ox.ac.uk`.

[2] Bruno Dufour, Christopher Goard, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, and Clark Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *Object-Oriented Programming, Systems, Languages and Architecture*, 2004. To appear.

[3] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, volume 2072, pages 327–355, 2001.

[4] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for Java. In *International Conference on Compiler Construction*, volume 2622, pages 138–152, 2003.

[5] Raja Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, McGill University, Canada, 2000.

Position paper for GPCE 2004 Software Transformation Systems Workshop

Anthony Sloane
Department of Computing, Macquarie University
asloane@ics.mq.edu.au
June 2004

An important class of generative programming systems contains those that employ cooperating generators to automatically build complex software from specifications.

For example, the Eli system, with which I have worked for more than 15 years, generates language processors from a variety of specification languages. Notations used include: regular expressions, context-free grammars, attribute grammars, symbol table descriptions, unparsing details, and structured output specifications. Eli-generated processors might be traditional compilers but, increasingly, Eli is being applied to related problems including domain-specific language implementation, construction of software analysis tools, and rapid application generation.

The main contribution of systems like Eli is to provide a "black box" experience so that the integration details of tools from different sources are hidden from the programmer who only needs to supply high-level specifications.

A major problem encountered when building a system like Eli that integrates many generators is how to describe the generation process by which:

1. Specifications are processed to obtain generator inputs.
2. Generators are invoked.
3. Outputs of generators are combined to form a processor.

As a simple example, Eli allows programmers to supply grammar specifications in EBNF notation. To avoid requiring the programmer to specify information twice, the grammar specification provides input to two generators: the grammar as a whole goes to a parser generator, and the literal terminal symbols are part of the input to a scanner generator. The scanner generator also receives a separate programmer-supplied specification of non-literal terminal symbols (such as integers or identifiers).

Thus, there is not necessarily a 1:1 correspondence between supplied specifications and generator inputs. A similar mismatch occurs at the output end. A particular generator may produce outputs that contribute to different components of the processor.

At present, Eli and other similar systems typically solve this integration process by using a build system such as Make to control the generation process. Eli uses Odin, an expert system for the automatic derivation of software artifacts. Odin is guided by a graph-based description of which artifacts can be constructed from which other artifacts. To actually carry out derivation steps, Odin invokes shell scripts that process specifications or run generators as required. Many shell scripts use standard Unix tools such as awk and sed to implement simple transformations; others use custom tools for more complex processing.

This approach works, but is somewhat unsatisfactory because it buries design decisions about how specifications are processed and how outputs are integrated in a variety of different places. These decisions may be distributed throughout the Odin derivation graph, shell scripts, awk or sed commands, or the code of custom tools. Comprehension for debugging and maintenance is hampered because the

input to a generator may come from a lengthy series of script or tool invocations. There is very little chance of analysing the formal properties of a generation process implemented in this way. Similar arguments apply to Make-based systems.

It seems that there is scope for software transformation to make a contribution to improving this situation. The integration problems at the input and output ends can be viewed as transformations from specifications to generator inputs, or from generator outputs to processor components. Having a holistic formal basis for these transformations that includes generator invocation would greatly assist debugging and maintenance of systems like Eli. Also, it would raise the generation process from one embodied by an ad-hoc implementation to a precise description that may be amenable to formal analysis and optimisation.

Software Transformation Systems

Douglas R. Smith
Kestrel Institute
Palo Alto, California

I have led projects that built a series of Software Transformation systems, including KIDS, Specware, Planware, Designware, and Epoxi.

KIDS (Kestrel Interactive Development System (1984-2000)) used a combination of constructive first-order inference, algorithm theories/tactics, and a variety of classic program transformations such as context-dependent simplification, finite differencing, and partial evaluation. Algorithm theories combine an algorithm template with axioms that are sufficient to guarantee the correctness of any instantiation. KIDS was used to synthesize many functional programs from input-output specifications, including a variety of complex high-performance scheduling applications.

The theory and practice of KIDS led to the development of a category-theoretic foundation in Specware (1994-present). Specifications higher-order logic are used to capture software requirements. Specification morphisms (and interpretations between theories) are used to represent specification refinement and speification structuring. Colimits exist in this category and are used to compose specifications. Similarly, library representations of design knowledge (e.g. algorithm theories, datatype refinements) are applied by means of colimits. Recently we have extended the specification language to express behavioral specifications in such as way that, again, morphisms serve for refinement and colimits compute compositions.

Several domain-specific extensions of Specware have been built so that users can automatically generate code without needing to know the theoretical underpinings of the system. Planware specializes in the synthesis of high-performance resource management systems. It uses state machines with constraints and service links to abstractly model both tasks and resources. In a typical application, a specification of 7 classes of resources is built up via both graphical and textual notations, roughly 1000 lines of specification text. From this input Planware automatically generates over 100,000 lines of CommonLisp code in a few minutes.

References

Smith, D.R., KIDS: A Semi-Automated Program Development System, IEEE Transactions on Software Engineering 16(9), Special Issue on Formal Methods, September 1990, 1024-1043.

Smith, D.R. and Lowry, M.R., Algorithm Theories and Design Tactics, Science of Computer Programming 14(2-3), Special Issue on the Mathematics of Program Construction, October 1990, 305-321.

Smith, D.R., Constructing Specification Morphisms, Journal of Symbolic Computation, Special Issue on Automatic Programming, Vol 16, No 5-6, 1993, 571-606.

Smith, D.R., Toward a Classification Approach to Design, invited paper in {\em Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology, AMAST'96}, LNCS 1101, Springer Verlag, 1996, 62--84.

Lee Blaine, Limei Gilham, Junbo Liu, Douglas R. Smith, and Stephen Westfold, Planware -- Domain-Specific Synthesis of High-performance

Schedulers, Proceedings of the Thirteenth Automated Software Engineering Conference, IEEE Computer Society Press, Los Alamitos, California, October, 1998, 270--280.

Smith, D.R., Designware: Software Development by Refinement, invited paper in Proceedings of the Eighth International Conference on Category Theory and Computer Science, Edinburgh, September, 1999.

Marcel Becker, Limei Gilham, Douglas R. Smith, Planware II: Synthesis of Schedulers for Complex Resource Systems, submitted for publication, 2004.

Douglas R. Smith, A Generative Approach to Aspect-Oriented Programming, to appear in Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE 04), Vancouver, October, 2004.

Specware, http://www.specware.org/

Generative Programming, Interface-Oriented Programming, and Source Transformation Systems

L. Robert Varney and D. Stott Parker
{varney,stott}@cs.ucla.edu

Abstract

We propose a new interface-oriented approach to generative programming and suggest that source-to-source transformations could play a complementary role in an interface-oriented programming system.

1 Introduction

Despite advances in programming language technology, large-scale software evolution and reuse remain challenging. One basic challenge is developing software libraries that encompass the right set of abstractions and associated implementation choices, providing interfaces that are both easy to use and efficiently implemented. This task is complicated by what Batory [BSST93] calls the feature combinatorics problem and Biggerstaff [Big94] calls the library scaling problem. These problems originate in programming languages that require implementations to be named where more abstract interfaces would suffice, creating *implementation bias*.

Biggerstaff, Batory, and others have suggested that the solution is not to stock libraries with prefabricated components covering all anticipated feature combinations, the solution is to factor abstractions and features into recomposable precursor components, and generate components with specific feature combinations as needed. *Generative systems*, described in section 2, do just that. However, there are important implications of splitting components and implementing them in abstract terms. One is that separation of concerns eventually requires some means for recomposition, and the other is that abstract implementations are difficult to implement efficiently out of context.

Generative systems require users to explicitly name the implementation fragments to be recomposed, transferring the implementation bias problem to users. *Interface-oriented programming* (IOP) [Var04], described in section 3, is an alternative programming language technology for separation of concerns that avoids implementation bias by relying on a generative mechanism called representation inference to automatically generate compositions of partial components. Although generative systems and IOP approach the integration of concerns differently, each encourages components to be implemented in abstract terms and is therefore susceptible to performance problems.

Efficient implementation of abstract programs without compromising abstractness has been the goal of source transformation systems. In source transformation systems, code is annotated with semantic information and separately defined transformations are applied to create a specialized, more efficient version of this abstract code. Although annotation and transformation can address the efficiency problem, they can also introduce yet another form of implementation bias, a form that manifests itself in the choice and ordering of particular transformations.

We believe that the strengths and weaknesses of IOP and source transformations may be complementary. IOP can be inefficient due to the use of high-level abstractions and needs a source of implementation variants, and source transformation systems suffer from implementation bias due to manual composition of specific transformations. In this paper we suggest that IOP and source transformation systems could be synergistically combined to solve both problems concurrently.

The paper is organized as follows. Section 2 reviews generative systems, section 3 discusses IOP, and the two are compared in section 4. We discuss software transformation systems in section 5, we suggest how IOP and source transformation systems might be combined in section 6, and we conclude the paper in section 7.

2 Generative Systems

We now discuss two generative systems, Batory's GenVoca model (and its successor, called AHEAD), and Czarnecki and Eisenecker's template metaprogramming approach called Generative Programming.

Batory and O'Malley [BO92] present the GenVoca model of software system design based on reuse of interchangeable components. In GenVoca, a component is a closely knit cluster of classes, and every component has both an abstract interface and a concrete implementation. Components implement abstract to concrete mappings by translating operations in the abstract interface to operations on lower-level objects in terms of their abstract interfaces. These lower level objects are provided to the component at instantiation time via parameters, and their implementations are unknown to the component.

These concepts can be modeled as a grammar or a set of equations, with abstract interfaces representing nonterminals and parameterized components representing the right-hand sides of productions. A special software tool called a layout editor is required to assist the user in composing valid compositions of components.

In later work, Batory and Geraci [BG97] describe in more detail how the GenVoca system validates compositions of parameterized components with associated design rules. More

recently the GenVoca model has been applied to the generation of development tool suites and has been reformulated as a new component model called AHEAD (Algebraic Hierarchical Equations for Application Design) [BSR03]. AHEAD extends the components-as-equations technique to other software artifacts, not just code.

Czarnecki and Eisenecker [CE99][CE00] present a component generation technique in which highly parameterized components are assembled into concrete configurations by a configuration generator, and configuration knowledge is stored apart from the components themselves. In this technique, called Generative Programming, implementation fragments are associated with abstract features, and the algorithms for combining features are expressed as template metaprograms that accept feature identifiers as arguments and compose the corresponding implementation fragments. The implementation of a particular feature combination can then be instantiated by invoking the template metaprogram with the corresponding feature identifiers, causing the template metaprogram to be specialized at compile-time to produce the desired code.

3 Interface-Oriented Programming

Interface-oriented programming [Var04] is an extension of object-oriented programming in which all program interdependencies are expressed via abstract interfaces. Current programming languages force specific implementations to be identified by name at points of instantiation dependency (dynamic object creation and inheritance), causing implementation bias. In IOP, instantiation dependencies can be expressed using abstract interfaces, and interfaces and implementations are strictly separated. As a result, inheritance in IOP is interface-oriented, supports both specialization and adaptation inheritance, and is decoupled from the implementation binding mechanism [VP04].

We are developing a programming language called ARC that implements these ideas. In ARC, a program is defined in terms of abstractions (interfaces) and partial representations. A partial representation implements a subset of one or more interfaces, and assumes the presence of implementations of one or more other interfaces. The assumed interfaces of a partial representation are not visible to clients of its interface - indeed, clients of an interface are never aware of the implementations they are using, and partial representations are never aware of the siblings they may coexist with in a complete representation. Partial representations are like mixins, but are never composed explicitly with other partial representations. Implementing IOP requires a new form of program generation we refer to as representation inference, in which partial representations are automatically assembled to construct complete representations that satisfy each partial representation's local constraints.

IOP allows an interface to be implemented many different ways. The primary source of variation comes from different implementation strategies that are explicitly captured in partial representations, and from the different ways in which partial representations may be combined to form alternative complete implementations. When using an interface, the user must assume that one of these complete implementations is chosen automatically and non-deterministically, although an extra-linguistic facility is provided to control the representation selection mechanism. We envision that this mechanism can be extended to allow dynamic evolution, but ultimately the representation choices are limited by the available handwritten partial representations and their automatically generated combinations.

4 Comparing IOP with Generative Systems

The main difference between IOP and the generative systems we have discussed is that IOP generates valid component compositions for a requested abstract interface automatically, whereas generative systems require partial implementation choices to be explicitly named at the site of an interface instantiation request.

GenVoca and AHEAD essentially provide high-level equations that drive code generators for parameterized components without support for inheritance. These code generators operate at the level of syntactic implementation fragments, fragments that do not distinguish between client-relevant details and internal implementation choices.

Czarnecki and Eisenecker's Generative Programming reduces the tedium associated with manual composition, but it is still necessary to manually compose specific combinations of implementation-oriented features. Also, each space of possible combinations requires its own template metafunction that must anticipate all of the different features that will be usable as arguments. Template metafunctions also do not distinguish between features that clients need to understand and features that are implementation details they should ignore. A feature is modeled as an implementation with some interface, but multiple implementations of the same interface are not considered.

In IOP each feature (partial representation) is defined locally and incrementally - everything we can say about how a feature may be combined with others is specified locally, the size of the specification depends only on what the feature depends on and not on the size of the total feature space, and the specification is monotonic with the addition of new features to the feature space (we do not have to revise our constraints with the addition of new features). In IOP the space of possible feature combinations is implicit in these constraints, whereas in GenVoca, AHEAD, and generative programming it must be explicitly managed as a whole.

5 Source Transformation Systems

Increasing the level of abstraction can improve development productivity and maintainability but can also cause runtime efficiency to deteriorate (Baage et al. [BKHV03]). Performance can be improved by specialized implementations, but specialized versions are more difficult to use correctly, and they tend to expose implementation details through interfaces, violating interface abstractions and entangling interfaces with implementations ([GL99]). Source transformation systems attempt to preserve the benefits of abstract program development with the need for efficiency by allowing source programs to be maintained manually in abstract form, and using automatic transformations to specialize them into more efficient forms.

Source to source transformations can improve performance but generally require manual insertion of semantic annotations and the selection and ordering of particular program transformations. Certain domain-specific optimizations are applicable only in certain contexts, and context is affected by the transformations themselves, thus transformation ordering is significant and opportunities for opti-

mization will depend on ordering ([BH03]). The challenge of composing the right transformations in the right order appears to be yet another instance of the tradeoff between abstraction and performance that the source transformation approach is designed to address – generic transformations are broadly applicable but slow, and specialized transformations are less reusable but fast – there is no free lunch. The problem is not source transformation per se, it may have more to do with how the transformations are chosen, and when.

Manual application of transformations not only fixes a choice of transforms (limiting its applicability), but is also error prone and may miss opportunities for optimization ([GL99]). Automation is needed not only to detect all optimization opportunities, it is also needed because the choice is not unique, and the set of applicable transformations may change as the nature of an application's workload changes.

6 Integrating IOP and STS

Although explicitly programmed variations of partial components are currently the primary source of raw material for IOP's representation inference mechanism, another possible source of variation could be source-to-source program transformations. Source transformation might transform partial components into new partial components, or could even transform complete representations. The result in each case is a larger population of candidate representations that the representation inference and selection mechanisms can choose from.

Conversely, the choice of what transformations to apply could be handled in a manner similar to how representations are inferred, yielding a *transformation inference* mechanism. Transformations should have known incremental effects on partial representations, and would have to be specified in terms of that abstract effect.

However, current source-transformation systems rely on program annotations that are not integrated with the type system, and require users to commit to a particular choice and ordering of transformations. For our suggestion to work, the semantic information needed by the source transformations should be integrated with the interface-oriented programming language, and the source transformations themselves must somehow be integrated with language's mechanism for implementation variation, essentially extending the representation inference mechanism to search not just for implementations that can be *assembled* from the available parts, but for implementations that can be generated by *transforming* the available parts.

7 Conclusion

In this paper we have suggested that the strengths and weaknesses of IOP and source transformations may be complementary, each solving a problem left open by the other. On the one hand, IOP solves the implementation bias problem but can suffer from the inefficiency of interface abstraction, and IOP also needs a fertile source of implementation variants. On the other hand, source transformation systems can automatically generate efficient new implementation variants but suffer from the implementation bias problem due to the need to manually commit to specific transformations. By combining the two mechanisms, IOP could use source transformations as a way to generate additional implementation variants, and particular combinations of interface-oriented partial transformations could be generated automatically by extending IOP's representation inference mechanism to perform transformation inference.

References

[BG97] D. Batory and B. J. Geraci. Component validation and subjectivity in genvoca generators. *IEEE Transactions on Software Engineering*, pages 67–82, 1997.

[BH03] O. S. Baage and M. Haveraaen. Domain-specific optimisation with user-defined rules in codeboost. In *Proceedings of the 4th International Workshop on Rule-Based Programming (RULE '03)*, volume 86 of *Electronic Notes in Theoretical Computer Science*. Elsevier, June 2003.

[Big94] T. J. Biggerstaff. The library scaling problem and the limits of concrete component reuse. In *Proceedings of the International Conference on Software Reuse*, pages 102–109, 1994.

[BKHV03] O. S. Baage, K. T. Kalleberg, M. Haveraaen, and E. Visser. Design of the codeboost transformation system for domain-specific optimisation of c++ programs. In *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 65–75. IEEE Computer Society Press, 2003.

[BO92] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM TOSEM*, 1(4):355–398, 1992.

[BSR03] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. In *Proceedings of International Conference on Software Engineering*, pages 187–197, 2003.

[BSST93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. In *Proceedings of ACM SIGSOFT*, pages 191–199, 1993.

[CE99] K. Czarnecki and U. W. Eisenecker. Synthesizing objects. In *Proceedings of ECOOP*, 1999.

[CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison Wesley, 2000.

[GL99] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Proceedings of DSL'99: The Second Conference on Domain Specific Languages*, Austin, Texas, 1999. The USENIX Association.

[Var04] L. Robert Varney. Interface-oriented programming. Technical Report TR-040016, UCLA Department of Computer Science, 2004.

[VP04] L. Robert Varney and D. Stott Parker. Inheritance decoupled: It's more than just specialization. In *Proceedings of ECOOP/MASPEGHI Workshop*, 2004.

Generation by Transformation in ASF+SDF

M.G.J. van den Brand and J.J. Vinju
Mark.van.den.Brand@cwi.nl, Jurgen.Vinju@cwi.nl

In this short paper we sketch two particular application areas of the ASF+SDF Meta-Environment: source-to-source transformations and code generation. Not surprisingly these two applications have much in common. ASF+SDF is composed of generic language technology. It offers the expressive power to cover these applications at a convenient level of abstraction.

Figure 1 summarizes an extreme viewpoint: source code related engineering activities can all be seen as language manipulations. Code generation and software transformation are represented by edges in this figure. Languages and tools that offer specific functionality for the easy definition of language translations are expected to perform well in both generative programming and software transformation. Several modern systems consciously implement this viewpoint, e.g. ASF+SDF, StrategoXT, and TXL.

Scannerless Generalized Parsing The syntax of both input and output formats are rigorously defined using the SDF formalism. We can describe the syntax of all formats that have a context-free grammar. From these descriptions parsers are generated that translate data from string format to tree format. The fact that we apply generalized parsing allows us to write the syntax definitions in a declarative way without worrying how to transform the grammar in a specific class, e.g. LL(k) or LR(k). Scannerless parsing loosens the restrictions that scanners usually impose on the input formats, which increases the number of programming languages that can be described using SDF. The generated parsers automatically construct maximally shared parse trees which contain all relevant grammatical information. Thus the resulting parse tree is fully typed with respect to the grammar.

Conditional Term Rewriting is an apt paradigm for tree deconstruction and construction. ASF is a term rewriting language with features such as matching and list matching, conditional rewrite rules, and traversal functions.

We use it to concisely describe transformations on parse trees. The parse trees produced by the parser are immediately processed by the rewrite engines, and the output of rewriting are fully typed parse trees again.There is no loss of information due to translation to some abstract term format. The type-safety of ASF specifications guarantees that the output parse trees correspond exactly to a syntax definition.

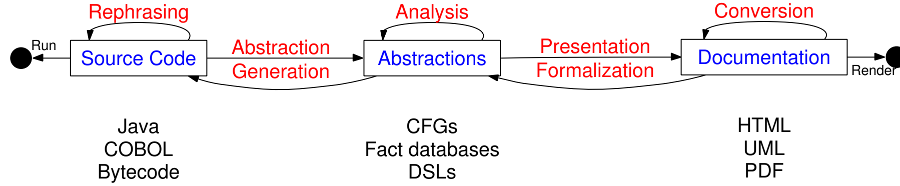


Figure 1: Three source-code representation tiers and their transitions.

The rewrite rules are written in the concrete syntax of the manipulated languages. Our formalism ASF is based completely on concrete syntax in contrast to other programming languages where this is an add-on. The fact that we perform rewriting on parse trees enables us to be "grammar-safe", and have access to a wealth of syntactical information such as layout, ambiguities, and position information.

Relation Calculus complements term rewriting by offering a programming interface on the level of fact analysis and manipulation. Logical operations and transitive closures can be expressed using term rewriting too, but on a less practical level of abstraction. In software transformation sometimes complex software analyses precede the actual transformation, and also in generative programming extensive analyses of the input data can improve the behaviour of the generator and the generated code.

Generic Pretty Printing is the final step for any transformation or generation process. The BOX language offers declarative primitives to render parse trees back to strings in a logical two-dimensional manner. Its implementations supports back-ends such as ASCII, HTML or LaTeX.

Several academical and industrial projects in software transformation and code generation have confirmed the aptitude of ASF+SDF Meta-Environment in these areas. ASF+SDF is used by an industrial partner (First Result B.V.) to translate information system specifications in a UML-like formalism into executable 3-tier architectures. In cooperation with the Vrije Universiteit Amsterdam advanced COBOL restructuring transformations are developed using ASF+SDF. The ASFC compiler, which translates ASF+SDF specifications into C code, is specified in ASF+SDF itself and bootstrapped. The first and the latter application are nice examples of powerful code generators. Two interesting question remain: is more specialized expressive power towards generative programming warranted? Does this specialization warrant research on other domain specific declarative languages?

At least the general, syntax oriented, type-safe and declarative features of the ASF+SDF Meta-Environment are a strong foundation for any source transformation or code generation project.

Reusable and Adaptable Strategies for Generative Programming

Position paper for the GPCE'04 Software Transformation Systems Workshop

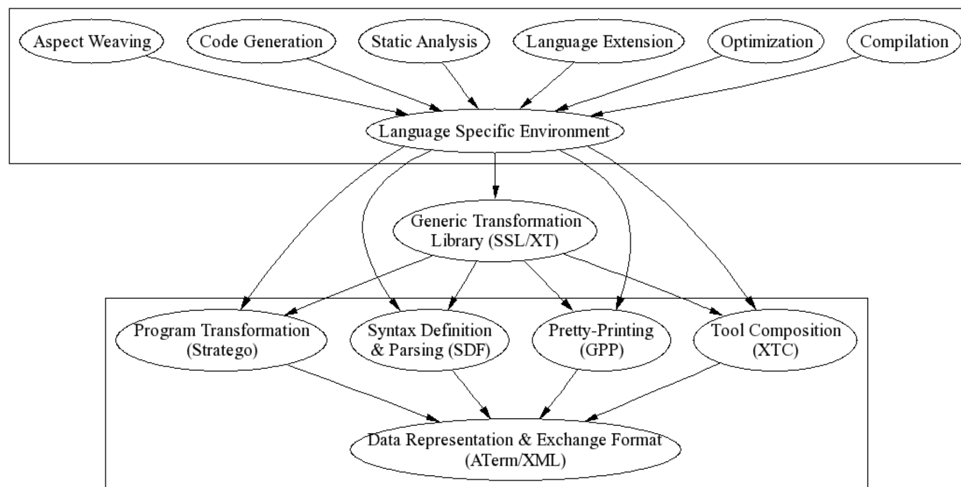
Martin Bravenboer and Eelco Visser

Institute of Information and Computing Sciences, Universiteit Utrecht, P.O. Box 80089 3508 TB Utrecht, The Netherlands {martin,visser}@cs.uu.nl

July 2004

Generative programming aims at increasing programmer productivity by automating programming tasks using some form of automatic program generation or transformation, such as code generation from a domain-specific language, aspect weaving, optimization, or specialization of a generic program to a particular context. Key for achieving this aim is the construction of *tools* that implement the automating transformations. If generative programming is to become a staple ingredient of the software engineering process, the construction of generative tools itself should be automated as much as possible. This requires an infrastructure with support for the common tasks in the construction of transformation systems. In the **Stratego/XT** project we have built a **generic infrastructure for program transformation** [2, 1]. In this position paper we give an outline of this infrastructure and indicate where we think the challenges for research and development of program transformations systems lie in the coming years.

Stratego/XTand its applications are organized in five layers (see diagram). **(1)** At the bottom layer is the **substrate** for a transformation system, that is the data representation and exchange format, for which we use the Annotated Term Format (ATerm) as basis, and XML where necessary to communicate with external tools. This substrate allows transformation systems to be rigorously componentized. **(2)** The **foundation** of any transformation system are syntax definition and parsing, pretty-printing, program transformation, and tool composition. The syntax definition formalism SDF provides modular syntax definition and parsing, supporting easy combination of languages. The pretty-printing package GPP supports rendering structured program



representations as program text. The program transformation language Stratego supports concise implementation of program transformations by means of rewrite rules and programmable strategies for control of their application. Finally, the XTC library supports composition of simple transformation tools into complex ones. These are all generic facilities that are needed in any transformation for any language. **(3)** In the middle is a library of transformations and transformation utilities that are not specific for a language, but not usable in all transformations either. The Stratego Standard Library (SSL) provides a host of generic rewriting strategies, and the XT toolset provides utilities for generating parts of a transformation system. **(4)** Near the top are specializations of the generic infrastructure to specific object languages. Such a **language specific environment** consists of a syntax definition for a language along with utilities such as semantic analysis, variable renaming, and module flattening. **(5)** Finally, at the top are the actual **generative tools** such as compilers, language extensions, static analysis tools, and aspect weavers. These tools are implemented as compositions of tools from the lower layers extended with one or more components implementing the specific transformation under consideration.

Where are we now in this development? The basic infrastructure (bottom two/three) layers is well established, readily available, and deployed in several projects. Although these components are gradually improved and extended, they form a reasonably stable development platform. The **next frontier** is the expansion of the infrastructure to generative applications. The tools in the top-level layer are the ones that matter since they are to be used by **application programmers**. Although we have also built a range of prototype language specific environments and generators to experiment with and validate our infrastructure, these do not yet form a product line by themselves. The main challenges to larger scale deployment are availability of *standard language specific components*, lack of documented *reusable strategies* at all levels of granularity, and an approach to make transformation *strategies adaptable*.

Although the languages and tools provided by Stratego/XT make the construction of transformation systems a lot easier than doing it from scratch using a general purpose programming language, it can still be a lot of work. Before a particular generative tool can be created, the hurdle of creating a language-specific environment must be taken. Therefore, the routine production of generative tools requires a library of **standard language specific components**, that is, front-ends for many standard and non-standard languages. The expectation that such front-ends will appear as spin-offs from production of specific generative tools is unjustified. The hardness and tediousness of this problem seems to be caused by the sheer complexity and irregularity of programming languages. Nonetheless, investigation of a higher-level and more declarative solution to the implementation of front-ends for real programming languages would we worthwhile.

Another challenge for transformation systems is to raise the level of reuse by providing **reusable strategies** at all levels of granularity, from micro-level transformations to macro-level tool compositions upto *process strategies*. As with all reuse, a sensible documentation and indexing of available solutions is key to benefitting from the available components. This is clearly a problem for new and even for more experienced users of Stratego/XT. With some 170 executable components and over 1200 strategy definitions in the library, it can be hard to locate the right ones for a specific transformation. While the available components form a very expressive programming environment, supporting the easy composition of all kinds of transformation systems, it requires knowledge of the available components and the ways to compose them. The cause of this problem is that the available components live at a lower level of abstraction than the problem being solved. To improve the productivity of meta-programmers we need higher-level **semantic strategies** that capture all aspects of some type of transformation, from the composition of tools to the traversals used in the actual transformation. Thus, one would expect reusable strategies for compilation, code generation, and aspect weaving, for example. For example, currently, the production of a code generator involves finding components for parsing, pre-processing, and pretty-printing and writing the actual generation strategy and rules. Often the code for such a generator will look very much like other generators, i.e. follow the same design pattern. A reusable code generation strategy should provide a standard composition and a pluggable traversal strategy that only needs to be instantiated with the name of the input and output languages and the generation rules. An ontology for such semantic strategies can then provide access to the components of the

infrastructure starting at the right level of abstraction.

It is not to be expected that such reusable strategies will be right for every possible situation. Therefore, these strategies should be **open and adaptable**. In the first place the strategies should be adaptable to the specific *languages* being transformed. In the second place strategies should be adaptable to the specific *program* that is being transformed or generated. Regarding the first point, languages differ in large and small ways from the standard model. A reusable strategy based on some general model of data-flow, say, needs to be adapted to the specific data-flow rules of the language under consideration. Regarding the second point: It is not uncommon that the programs produced by generators, especially those that generate high-level programs, are further edited by application programmers in order to fit the generated code in their application. Generators that produce code templates even assume this mode of work. Of course, this produces a maintenance nightmare; when the source of generation changes, all modified files need to be merged with the newly generated ones, undoing all benefits of generation. Thus, it should be possible to adapt the product of generation without physically modifying it, either by means of input to the generator, by generating code that can be adapted (e.g. through subclassing), or by applying further transformations to the generated code (e.g. using aspects).

References

[1] `http://www.stratego-language.org`.

[2] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

Transforming Object-Oriented Programs into Structurally Reusable Components for Generative Reuse

Hironori Washizaki[1] and Yoshiaki Fukazawa[2]

[1]*National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan*, *washizaki@acm.org*

[2]*Waseda University, 3-4-1 Okubo, Shinjuku-ku, Tokyo, Japan*, *fukazawa@waseda.jp*

Abstract

We propose a technique for transforming part of the object-oriented programs into structurally reusable components by our new refactoring "Extract Component" to realize the generative reuse of the existing programs. Our refactoring can identify and extract components composed of classes from existing OO programs, and modify the surrounding parts of extracted components in original programs. We have developed a tool that performs our technique automatically and extracts JavaBeans components from Java programs.

1. Introduction

Since object-oriented (OO) classes usually have complex mutual dependencies, it is difficult to reuse parts of existing OO programs composed of classes rapidly and effectively. If a significant function is realized by a set of classes, programmers who want to reuse the function must examine the dependencies among related classes and acquire all depending classes. Since such manual examination activities entail a high cost for programmers, the merits of reuse might be reduced or lost.

In this paper, we propose a technique for identifying structurally reusable candidate parts of OO programs and transforming these parts into JavaBeans[1] components automatically by our new refactoring, "Extract Component[2]." Our technique targets Java language as the OO programming language and JavaBeans as the fundamental component architecture.

2. Component Extraction

To extract reusable components from Java programs, we first define a class relation graph (CRG) that represents the relations among classes/interfaces in the target Java program. In CRG, each node denotes a programmer-made Java class or interface. Each edge denotes a relation (inheritance, instantiation, and reference) between two classes/interfaces. The CRG is obtained by a static analysis for the target program. Figure 1 shows an example Java code and its CRG. Details of the CRG's definition are shown in [2].

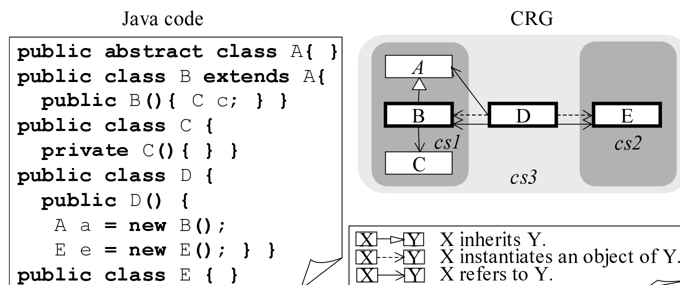


Figure 1. Program code and its CRG

Next, we can automatically detect all possible clusters on the CRG by analyzing reachabilities regarding relations among nodes. The cluster is a set of classes/interfaces that satisfies the following two requirements, and is denoted as $cs = (c_f, V_c)$, where c_f is the Facade class and V_c is a set of all classes/interfaces in the cluster. Each cluster becomes a candidate of a structural reusable component

- The set includes one Facade class, which plays the role of the facade for the outside of the classes/interfaces based on the Facade pattern[3]. The Facade class must have one public default constructor (a constructor without any arguments).
- All classes/interfaces necessary for instantiating an object of the Facade class are packaged into the same set.

For example, we obtained the following three clusters on the CRG in Figure 1: $cs_1 = (\mathtt{B}, \{\mathtt{A,\ B,\ C}\})$, $cs_2 = (\mathtt{E}, \{\mathtt{E}\})$, and $cs_3 = (\mathtt{D}, \{\mathtt{A,\ B,\ C,\ D,\ E}\})$.

After detection, we can produce a new Facade interface to enable the use of a cluster from the outside, and compile all related source codes of the cluster into the form of a Java archive (JAR) file including one JavaBeans component. The acquired component does not always represent a semantically reusable component in possible contexts. However, the acquired component is the structurally reusable component, because the internal structure is hidden from outside, and the component has no dependency on elements outside.

When extracting components corresponding to specified clusters, the surrounding parts of clusters should use newly extracted components in order to avoid the situation where two sets of classes that provide the same function exist in the same library. This modification is a kind of refactoring[4] because this modification does not change the observable behavior of the original program. We call this refactoring, "Extract Component," and the key steps of this refactoring are shown below.

(1) Obtain all clusters and select one cluster $cs = (c_f, V_c)$.
(2) Create a new Facade interface i_f.
(3) Add the declarations of all public methods implemented within c_f into i_f.
(4) Add the declarations of the setter methods and getter methods corresponding to all public fields of V_c into i_f. Add the implementations of the setter methods and getter methods into c_f. This step is a kind of Encapsulate Field refactoring[4].
(5) Compile and package all classes/interfaces in V_c and i_f into one JAR file.
(6) Change the program codes, which assign new values to fields of V_c (or refer the values of fields of V_c) in outside of the cluster, to the program codes that invoke the setter methods (or getter methods) of i_f.
(7) If the implicit widening reference conversion from c_f to another class/interface c_e in V_c exists in outside of the cluster, insert the explicit reference conversion program code, which converts the reference types from c_f to c_e, into all parts where the implicit conversion exists.
(8) Change the reference types, which refer to c_f in outside of the cluster, to those that refer to i_f.

3. Automated Tool

We have developed a tool that analyzes Java program source code, displays CRG, and performs automatically all necessary steps of the Extract Component refactoring. Since our tool can be treated as a generator of reusable components, it is thought that our tool realizes the generative reuse[5] of the existing Java programs.

Using our tool, we have attempted to extract all components from a Java class library KFC [6] (number of all classes/interfaces: 224). 13 components have been automatically extracted from the library. Names of the extracted components' Facade class are `BasicPSModifier`, `BasicTSModifier`, `DefaultHTMLReaderTarget`, `VDashedBorder`, `DTD`, `DefaultCompareAdapter`, `ColorButton`, `BorderedPane`, `Panel`, `CompositeKeyAction`, `Keymap`, `TextCaret`, and `ConcreteTextConstraint`. Many of them are components with high generality, such as Button and Panel. These components can be reused independently in other programs. It is found that the Extract Component refactoring is useful for extracting components from a class library.

4. Application and Discussion

Using the Extract Component Refactoring tool, we are currently developing a component-extraction-based program search system[2]. Our system extracts structually reusable components from a collection of Java programs, and generates indexes for newly extracted components. In our system, the extracted components can be searched by keywords via a web browser.

One of important issues for realizing the generative reuse for the existing programs is how to customize the extracted components to match the new application contexts. We believe that combining traditional generative techniques (such as the template programming) with our technique will be helpful to overcome this issue.

References

[1] Hamilton, G. JavaBeans 1.01 Specification (1997).

[2] Washizaki, H. and Fukazawa, Y. Component-Extraction-based Search System for Object-Oriented Programs, 8th International Conference on Software Reuse, LNCS Vol.3107 (2004).

[3] Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley (1994).

[4] Fowler, M. Refactoring: Improving the Design of Existing Code, Addison-Wesley (1999).

[5] Biggerstaff, T.J. A Perspetive of Generative Reuse, Annals of Software Engineering, Vol.5, No.1 (1998).

[6] Yasumatsu, K. KFC, http://openlab.jp/kyasu/

Transformation Systems for DSLs, Architectural Styles, and Graphical Languages

David S. Wile
Teknowledge Corp.
dwile@teknowledge.com

My position is simply the observation that *all of the transformation-based tools built for general-purpose languages in the last 25 years have analogous counterparts for manipulating Domain Specific Languages, Architectural Styles, and Graphical Idioms.* In particular, each of the following is germane:

- Language Specification – a set of constraints or templates to restrict designs
- Parsing – adherence of a design to the language specification
- Syntax-Direction – automated aid to constructing specifications that adhere
- Type Checking – imposing uniformly a set of more global constraints beyond the (generally local) syntactic constraints
- Abstract Syntax – an intermediate representation capturing the essential concepts of the domain
- Semantics Specification Mechanisms and Issues
    - Attribute Grammars
    - Transformations – within a language
    - Translations – between languages
    - (Other) Homomorphisms – into algebraically similar structures
    - Establishing transformation validity
- Traversal Mechanisms
    - Metaprogramming Calculi – programs as data
    - Strategies – heuristics for transformation
    - Visitor patterns – a calculus for OO representations of AST transformations
- Debugging Aids – errors related to source specifications and data structures
- Re-engineering aids (inverses for each semantics specification mechanism)
- Design recording aids
    - Historical Development – keeping track of a design history
    - Pedagogical Development – when can a design history be replayed?
    - Requirements-based Development – why are things as they are?

It is not the case that the analogies are always straightforward. For example, the visitor pattern does not carry over directly into the domain of graphical idioms, where the existence of potential cycles requires one to consider what happens when a node is "revisited." Nonetheless, researchers are certain to reinvent each of these concepts for the current terms without being aware of what has gone before, unless these similarities are raised explicitly to the relevant research communities.

I think a valuable contribution of this workshop would be a compilation of important articles on these topics, circulated via a web site or a mailing list to the attendees. To illustrate the depth of the heritage of papers on these topics I include the following fairly random smattering of my own and my close colleagues' *early* papers:

1. Balzer, R. M., Goldman, N. M. and Wile, D. S. On the transformational implementation approach to programming. In Proceedings of the Second International Conference on Software Engineering, October, 1976. Pp. 337-344.
2. Balzer, R, Goldman, N. Principles of good software specification and their implications for specification languages. *Specification of Reliable Software:* IEEE Computer Society. 1979. Pp. 58-67.
3. Balzer, R.M., Cheatham, T.E., and Green, C. Software Technology in the 1990's: Using a New Paradigm. *Computer Magazine,* 1983.
4. Bauer, F. L. Programming as an evolutionary process. *Proceedings of the Second International Conference on Software Engineering*, San Francisco, California: IEEE. October, 1976. Pp.223-234.
5. Bauer, F.L., Broy, M., Partsch, H., Pepper, P. and Wirsing, M. The Munich Project CIP. Vol. I: The Wide Spectrum Language CIP-L. *Lecture Notes in Computer Science 183,* Springer Verlag, Berlin. 1985.
6. Boyle, James M. Program adaption and transformation. In *Practice in Software Adaption and Maintenance*: Proceedings, Workshop on Software Adaption and Maintenance, Berlin, North-Holland, 1979. Pp. 3-20.
7. Burstall, R.M. and Darlington, J. A transformation system for developing recursive programs. *JACM* 24:1, 1977, pp. 44-67.
8. Cheatham, T.E. Jr., Holloway, G.H., and Townley, J.A., Program refinement by transformation. In Proceedings of the Fifth International Conference on Software Engineering, San Diego, CA March, 1981, pp. 430-437.
9. Donzeau-Gouge, V., Kahn, G., Lang, B., and Mélèse, B. “Document Structure and Modularity in Mentor,” *Proceedings of the ACM SIGSOFT/SIGPLAN Software Symposium on Practical Software Development Environments* (1984), pp. 141-148.
10. Feather, M.S. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems* 4(1), 1982 pp. 1-20.
11. Feather, M.S. A survey and classification of some program transformation approaches and techniques. Meertens, L.G.L.T. ed. *Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation, Bad Toelz,* North-Holland. 1986. 165-195.
12. Gerhart, S. L. Correctness preserving program transformations. In Proceeedings, 2nd ACM POPL Symposium, Palo Alto, CA. 1975. Pp. 54-66.
13. Green, C.C. and Barstow, D. On program synthesis knowledge. *Artificial Intelligence* 10, 1978, pp. 241-279.
14. Kant, E. On the efficient synthesis of efficient programs. *Artificial Intelligence* 20, 1983, pp. 253-305.
15. Kibler, D.F. *Power, Efficiency, and Correctness of Transformation Systems*, PhD thesis, University of California at Irvine, 1978.
16. Neighbors, J. *Software Construction Using Components*. Ph.D. Thesis. Computer Science Department. University Of California, Irvine, 1981.
17. Paige, R., Transformational programming – applications to algorithms and systems. In *Proceedings of the 10th ACM POPL Symposium*, Austin, TX. 1983, pp. 73-87.
18. Pepper, P., ed. *Program Transformation and Programming Environments* NATO ASI Series F: Computer and Systems Sciences. 8. Springer Verlag, 1983.
19. Reps, T. W. and Teitelbaum, T. *The Synthesizer Generator.* Springer-Verlag, New York (1988).
20. Rich, G. A formal representation for plans in the Programmer’s Apprentice. In Proceedings, 7th Interntaional Joint Conference on Artificial Intelligence. 1981.
21. Scherlis, W.L. Program improvement by internal specialization. In Proceedings, 8th ACM Symposium on the Principles of Programming Languages, Williamsburg, VA. January, 1981. Pp. 41-49.
22. Wile, D. S. Type transformations. *IEEE Transactions on Software Engineering.* 1(SE-7):1981. Pp. 32-39
23. Wile, David S. Program Developments: Formal Explanations of Implementations. *Communications of the ACM*, 26:11 (1983).
24. Wile, D.S. Local Formalisms: Widening the Spectrum of Wide-Spectrum Languages. Meertens, L.G.L.T. ed. *Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation, Bad Toelz,* North-Holland. 1986. 459-481.

Semantic Analysis in Software Transformation

Eric Van Wyk and Eric Johnson
University of Minnesota

October 24, 2004

When transforming a source program or specification there are two important questions that must be answered:

1. What constructs in the source need to be transformed? and
2. What should the selected constructs be transformed into?

It is our position that using semantic analysis of the source program in answering these two questions leads to powerful transformational systems. Our main interest is in designing extensible compiler frameworks that allow programmers to easily import into their "host language" compiler, a unique set of domain-specific language extensions suitable for their task at hand. (They must be able to do this with no implementation-level knowledge of the extensions.) These constructs need to be as well-developed as the host language's native constructs. Thus, language feature designers should be able to specify new language constructs together with both their semantic analyses and their optimizing transformations.

In term rewriting systems, only the syntactic structure of the source is examined to determine the constructs to transform. While this is all that is needed in many domains, it does limit the transformation system. For example, transformations based on an expression's type are often useful. Semantic analysis is also helpful in determining what a construct is translated into. Below, we describe an example from computational geometry (CG) in which exact precision numeric types are transformed into arrays of fixed precision types. Since exact precision values are used in a rather limited way in CG programs, semantic analyses are used in generating highly efficient implementations of their arithmetic operations.

There are many well-understood techniques for transformation via rewriting (Baader and Nipkow, 1998; Visser, 2001) based on declarative specifications of rewrite rules. These generally do not take into account semantic information of the source; (Lacey and de Moor, 2001) is one exception. Attribute grammars are a well-understood mechanism for specify semantic analyses of context free languages (Knuth, 1968). With the addition of higher-order attributes, new attributed trees can be constructed and thus transformational systems can be built that make use of semantic analyses. However, higher order attribute grammar systems do not support the modularity that we seek (Van Wyk et al., 2002). If the host language and the language extensions are specified as higher order attribute grammars, the programmer must understand their implementation and write attribute definitions in order "glue" the extensions into the host language attribute grammar.

An extension to attribute grammars, called "forwarding" (Van Wyk et al., 2002), solves this problem as it allows us to mimic simple rewriting inside an attribute grammar framework. We have built a prototype extensible language framework based on forwarding attribute grammars in which the host language and the language extensions are specified as attribute grammar fragments. To use forwarding, the production defining a new language construct in a language extension will generate a semantically equivalent construct in the host language. This semantically equivalent construct represents its implementation in the host language and provides definitions to attributes that are not explicitly specified (by attribute definition rules) on the production for new language construct. If a node in the attributed syntax tree is queried for an attribute

for which its does not have an explicit attribute definition, it "forwards" that query to the semantically equivalent construct which provides the answer (either directly or by forwarding again).

In the domain of computational geometry, we have implemented a number of extensions that help geometers write robust CG programs. It is common practice in this domain to base algorithms on a "geometric primitives". These are expressions that that return a qualitative, not quantitative, result about a relationship between geometric objects - e.g. is a given point inside a given circle. It is only in these primitive where geometric entities are compared or examined. One extension we have implemented provides an implementation of exact-precision integers (Fortune and van Wyk, 1996) that are used in the geometric primitives where the number of bits needed to store intermediate results may exceed that supported by the hardware. We use static analysis of the exact expressions so that the arithmetic operations on exact types can generate a highly optimized semantically equivalent host language construct to forward to. Because exact-precision types are used only in geometric primitive expressions and not in variables whose value may be assigned inside of branch or loop statements, a number of static analyses are possible. For example, the number of bits that will be required to store the intermediate exact-precision value can be statically determined. Thus the loops that run over arrays of fixed-precision numbers (representing exact-precision values) have statically known bounds and can be unrolled thus removing the looping overhead. Since computational geometry programs spend a considerable amount of time executing these primitives, this provides for a significant speed up.

In our experience, code transformation assisted by semantic analysis is especially expressive. It is the combination of semantic analysis of attribute grammars and the transformation mechanism implemented by forwarding the allows us to design highly modular language specifications and thus highly modular extensible compiler frameworks that allow programmers to easily import expressive and efficient new language features.

References

Baader, F. and Nipkow, T. (1998). *Term Rewriting and All That.* Cambridge University Press, Cambridge, U.K.

Fortune, S. and van Wyk, C. J. (1996). Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248.

Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145. Corrections in 5(2):95–96, 1971.

Lacey, D. and de Moor, O. (2001). Imperative program transformation by rewriting. In *Proc. 10th International Conf. on Compiler Construction*, volume 2027 of *Lecture Notes in Computer Science*, pages 52–68. Springer-Verlag.

Van Wyk, E., de Moor, O., Backhouse, K., and Kwiatkowski, P. (2002). Forwarding in attribute grammars for modular language design. In *Proc. 11th International Conf. on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 128–142. Springer-Verlag.

Visser, E. (2001). Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In Middeldorp, A., editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag.