Invited Application Paper

Re-Engineering C++ Components Via Automatic Program Transformation

Robert L. Akers, Ph.D.
lakers@semdesigns.com

Ira D. Baxter, Ph.D.
idbaxter@semdesigns.com

Michael Mehlich, Ph.D.
mmehlich@semdesigns.com

Semantic Designs Inc.
12636 Research Blvd. #C214
Austin, Texas, USA 78759-2200
512-250-1018

ABSTRACT

Automated program transformation holds promise for a variety of software life cycle endeavors, particularly where the size of legacy systems makes code analysis, re-engineering, and evolution very difficult and expensive. But constructing transformation tools that handle the full generality of modern languages and that scale to very large applications is itself a painstaking and expensive process. This cost can be managed by developing a common transformation system infrastructure that is re-used by an array of derived tools that each address specific tasks, thus leveraging the infrastructure cost over the various tools.

This talk describes DMS[1], a practical, commercial program analysis and transformation system, and discusses how its infrastructure was employed to construct the Boeing Migration Tool (BMT), a custom component modernization application being applied to a large C++ industrial avionics system. The BMT automatically transforms components developed under a 1990's era component style to a more modern CORBA-like component framework, preserving functionality. We describe the DMS infrastructure and the BMT application itself, illustrating some of the kinds of syntheses and transformations required and some of the issues involved with transforming industrial C++ code. We also discuss the development experience, including the strategies for approaching the scale of the migration, the style of interaction that evolved between the tool-building company and its industrial customer, and how the project adapted to changing requirements.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming – *Automating analysis of algorithms, Program Modification, Program Synthesis, Program Transformations.* D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Computer-aided software engineering (CASE).* D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement – *Restructuring, reverse engineering, and reengineering.* D.2.13 [**Software Engineering**]: Reusable Software – *domain engineering.* D.3.4 [**Programming Languages**]: Processors – *Parsing, Translator writing systems and compiler generators, Code Generation.*

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 Software Re-Engineering Toolkit

DMS provides an infrastructure for software transformation based on deep semantic understanding of programs. Programs are internalized via DMS-generated parsers that exist for virtually all conventional languages. 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 has been under development for 9 years, and 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 includes the following tightly integrated facilities:

- A hypergraph foundation for capturing program representations (e.g., ASTs, flow graphs, etc.) in a form convenient for processing.
- Complete interfaces for procedurally manipulating general hypergraphs and ASTs.

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

PEPM'04, August 24-25, 2004, Verona, Italy.

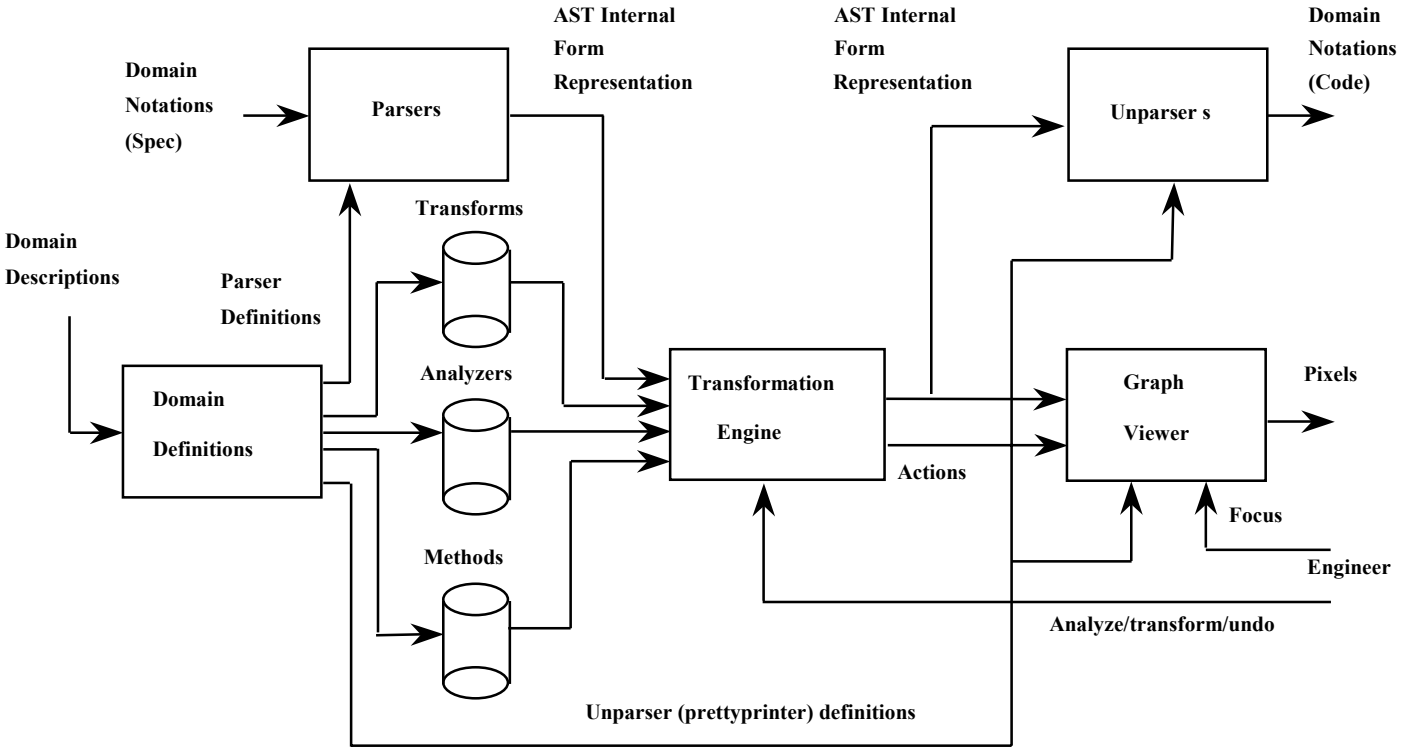

Figure 1: DMS Architecture

- A means for defining language syntax and deriving parsers and prettyprinters for arbitrary context free languages to convert domain instances (e.g. source code) to and from internal forms.
- Support for defining and updating arbitrary namespaces containing name/type/location information with arbitrary scoping rules, and support for name and type analysis.
- An attribute evaluation system for encoding arbitrary analyses over ASTs.
- An AST-to-AST rewriting engine that understands algebraic properties (e.g., associativity and commutativity).
- The ability to specify and apply source-to-source program transformations based on language syntax. Such transforms can operate within a language or across language boundaries.
- A procedural framework for connecting these pieces and adding arbitrary code.

The DMS architecture is illustrated above. Notice that the infrastructure supports multiple domain notations (source code languages), so that multiple languages can be handled or generated by a given tool.

We are presently implementing a general scheme for capturing arbitrary control flow graphs (including exceptions, continuations, parallelism and asynchrony) and carrying out data flow analyses across such graphs. Our goal is to build scalable infrastructure. One aspect is support for computational scale, which is addressed by implementing DMS in a parallel programming language, PARLANSE [1], enabling DMS to run on commodity x86 symmetric-multiprocessing workstations.

C++ is among the many domains implemented within DMS, and the system contains complete preprocessors, parsers, name and type resolvers, and prettyprinters 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 with their preprocessor directives preserved, even in the presence of preprocessor conditionals.

DMS as presently constituted 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 implementing tasks such as dead and clone code elimination, test code coverage, execution profiling, source code browsing, and static metrics analysis.

A more complete discussion of DMS is presented in [2]. DMS-based tools are described on the Semantic Designs web page [3].

2. THE BOEING MIGRATION TOOL

Boeing's Bold Stroke avionics component software architecture is based on the best practices of the mid 1990's [4]. Component technology has since matured, and the CORBA component model has emerged as a standard. The U.S. Government's DARPA-PCES program and OMG are sponsoring development of a CORBA-inspired standard real time embedded system component model [5], which offers standardization, superior encapsulation and interfaces for ongoing development of distributed, real time, embedded systems such as Bold Stroke. This standardization also provides a base for tools for design and analysis of such systems. Boeing wishes to upgrade its software to a more modern architecture, a proprietary CCMRT variant known as PRiSm. The

task of converting components is straightforward and now well understood, but a great deal of detail must be managed with rigorous regularity and completeness. Since Bold Stroke is implemented in C++, the complexity of the language and its preprocessor requires careful attention to semantic detail. With thousands of legacy components now fielded, the sheer size of the migration task is an extraordinary barrier to success. With the use of C++ libraries, approximately 150,000 lines of C++ source contributes to a typical component, and a sound understanding of the component's name space requires comprehension of all this code.

To deal with the scale, semantic sensitivity, and regularity issues, DARPA, Boeing, and Semantic Designs (SD) decided on an automated approach to component migration using a DMS-based tool. DMS, with its C++ front end complete with name and type resolution, its unique C++ preprocessor, which allows both the expansion (for understanding) and the preservation (for source code re-creation) of preprocessor directives, its transformation capability, and its scalability, was uniquely qualified as a substrate for constructing a migration tool. Automating the migration process assures regularity of the transformation across all components and allows the examination of transformation correctness to focus primarily on the general transforms rather than on particular examples that may be idiosyncratic.

The legacy component structure was essentially flat, with all the methods contributing to a component collected in a very few classes (often just one), each defined with `.h` and `.cpp` files. One principal piece of the migration involves factoring a component into facets, which would form distinct classes reflecting different areas of concern. Some facets encapsulate various functional aspects and are specific to each component. Others capture protocols for inter-component communication; while these protocols are common in style among all components, code specifics vary with the components' functional interfaces.

Factoring a component into functional facets requires human understanding. Essentially, the legacy interface methods must be sorted into bins corresponding to the facets, and indicative names given to the new facet classes. 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 names the facets and uniquely identifies which methods (via simple name, qualified name, or signature if necessary) comprise its interface. The bulk of the migration engineer's task is the formulation of facet specifications for all the components to be migrated.

The BMT translates components one at a time. Input consists of the source code, the facet specification for the component being translated, and the facet specifications of all components with which it communicates, plus a few bookkeeping directives. Conversion-related input is succinct.

The facet language itself is defined as a DMS domain, allowing DMS to automatically generate a parser from its grammar. (The BMT therefore is a multi-domain application, employing both Visual C++ and the facet language.) 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.

After processing the facet specifications, the BMT parses and does full name and type resolution on the C++ source code base, including files included by any of the components in play. The name resolver constructs a symbol table for the entire base, allowing lookup of identifiers and methods with respect to any lexical scope within the source code base. Only by internalizing the entire problem in this manner can symbol lookups and the transformations depending on them be guaranteed sound. This is one key point that defeats scripting languages as C++ transformers. Three particular transformations typify what the BMT does to perform the component migration:

- New classes for facets and their interfaces are generated based on the facet specifications. One incarnation of the BMT generates a base class for each facet that is essentially a standard form. A "wrapper" class is also generated, inheriting from the facet, and containing one method for each method in the functional facet's interface. The wrapper methods simply relay calls to the appropriate method in the component's core classes. Constructing the wrapper methods involves replicating each method's header and utilizing its arguments in the relayed call. Appropriate `#include` directives must be generated for access to entities incorporated for these purposes, as well as for standard component infrastructure. A nest of constructor patterns expressed in the DMS pattern language are used to pull the pieces together into a class definition. After constructing the facets and wrappers, the BMT must then transform all the legacy code calls to any of the facets' methods, redirecting original method calls to the core class to instead call the appropriate wrapper method via newly declared pointers. This is done using source-to-source transforms with conditionals to focus their applicability.

- Newly generated "receptacle" classes provide an image of the outgoing interface of a component to the other components whose methods it calls. Since a particular component's connectivity to other components is not known at compile time, the receptacles provide a wiring harness through which dynamic configuration code can connect instances into a flight configuration. Constructing the receptacles involves searching all of a component's classes for outgoing calls and generating code to serve each connection accordingly.

- Event sinks are classes that represent an entry point through which an event service can deliver its product. Since the code for event processing already exists in the legacy classes (though its location is not specified to the BMT), synthesizing event sinks involves having the BMT identify idiomatic legacy code by matching against DMS patterns for those idioms. Code thus identified is moved into the new event sink class, which is synthesized with a framework of constructor patterns. Definitions and `#include` directives supporting the moved code must also be constructed in the event sink class.

3. EXPERIENCE

The project is still in progress, but we can make a number of observations.

The customer, Boeing, has extensive expertise in avionics and component engineering, but only a nascent appreciation of transformation technology. The tool builder, Semantic Designs, understands transformation and the mechanized semantics of C++, but had only cursory prior understanding of CORBA component technology and avionics. Furthermore, customer and tool builder are geographically separated.

As it turns out, this all had its benefits, leading to a clean factoring of roles that forced clarity at the operational boundaries. Once a basic understanding of the component structure was communicated, Boeing chose a particular component to use as a foil for advancing the work. They performed a hand conversion which served to force details into consideration and helped them solidify the target via experimentation, while giving SD a concrete image of the target and a benchmark for progress. Being unburdened by application knowledge, SD was able to ask questions that focused purely on translation issues, removing from the conversion endeavor the temptation to make application-related adjustments that could add instability.

With the task cleanly factored, a mode of electronic communication evolved that reduced the need for travel and staff commitments to meetings. The hand-translated component was used for periodic benchmarking and evaluation, and its elaboration served to communicate requirements changes. SD periodically shipped the results of the auto-conversion and versions of the tool itself to Boeing for evaluation, and Boeing made suggestions with respect to the benchmark results. This spared Boeing engineers the difficulty of evaluating transforms abstractly. While they developed an appreciation for the capabilities and limits of the technology, they did not need to spend time learning to converse in terms of the transformation rules themselves.

SD had to do its development without access to Boeing's sensitive proprietary code base. Boeing partly finessed this potential problem by using a non-sensitive trial component. But lack of access to the full source code base forced the tool builders to prepare for worst case scenarios of what C++ features may be encountered by the BMT. This had the desirable effect of forcing development of our C++ preprocessing infrastructure to handle the cross product of preprocessing conditionals, templates, and macros. These improvements both harden the tool against unanticipated stress and strengthen the DMS infrastructure for future projects.

Various factors forced major changes of direction during the project. Initially, the target component model was essentially a pure CORBA structure, since the details of the real-time avionics variant had not thoroughly communicated to SD. A partial prototype translation was implemented on this basis. Then came a midstream decision to move first to a wrapper approach, which would have resulted in throwing away a lot of hand-translated code, had that been the mode of operation. The impact on the BMT approach was quite limited, though, since the change required only adjustments to the generative patterns and some organizational code, not to the tool structure as a whole. Likewise, a switch back to the fully defined PRiSm component model is anticipated at a date which would have come too late in a hand translation effort, but which the automatic conversion can accommodate. Re-tooling the BMT will require less effort, and once this is done, all components can be re-translated as easily as one.

Essentially, the role of SD in this effort was to develop a custom migration tool on top of our DMS infrastructure, while Boeing's task was to provide requirements and become the end user of the tool. There is no reason in principle that Boeing could not have been the tool developer. We encourage our users to consider taking our infrastructure and classes we offer, and learning to build tools themselves. The experience gained in building one tool leverages quickly into the ability to create new tools when the need arises, with the long-term effect being the incorporation of transformational methods into the users' software development culture. For an organization that requires only a single DMS-based application, the cost of training and learning by hard experience are not justified. SD's experienced engineers can deliver a first product more quickly. But DMS is designed to be distributed as a product, and we encourage organizations whose applications are so strictly proprietary that outsourced tool development is inappropriate and organizations which can envision the long-term benefits of adopting the DMS custom tool development methodology to take the toolkit and train to become DMS engineers. Academic research environments also seem particular open to this endeavor, as they can leverage the DMS infrastructure to allow researchers to focus directly on their problems of interest rather than building and maintaining a tool-building environment or hacking tools together from inadequate pieces.

A few over-arching observations apply to this and other mass transformation projects:

- *Mass migrations are best not mingled with changes in business logic, optimization, or other software enhancements*. Entangling tasks muddies requirements, induces extra interaction between tool builders and application specialists, and makes evaluation difficult, at the expense of time and money. Related tasks may be considered independently, applying new transformation tools if appropriate.

- *Automating a transformation task helps deal with changing requirements*. Modifying a few rewrite rules, constructive patterns, and organizational code is far easier and results in a more consistent product than revising a mass of hand-translated code. Changes implemented in the tool may manifest in all previously migrated code by simply re-running the modified tool on the original sources. This allows blending the requirements definition timeframe into the implementation timeframe, which can significantly shorten the whole project.

- *Cleanly factoring a migration task between tool builders and application specialists allows proprietary information to remain within the customer's organization while forcing tool builders toward optimal generality*. Lack of access to proprietary sources, or in general lack of full visibility into a customer's project induces transformation engineers to anticipate problems and confront them in advance by building robust tools. Surprises therefore tend to be less overwhelming.

- *Automated transformation allows the code base to evolve independently during the migration tool development effort*. To get a final product, the tool may be re-run on the most recent source code base at the end of development. There is no need for parallel maintenance of both the fielded system and the system being migrated.

- *Using a mature infrastructure makes the construction of transformation-based tools not just economically viable, but advantageous*. Not doing this is infeasible. Language front ends and analyzers, transformation engines, and other components are all very significant pieces of software. The BMT contains approximately 1.5 million lines of source code, but most is infrastructure. Only 11K lines of code are BMT-specific. Furthermore, off-the-shelf components are inadequate to the task. For example, lex and yacc do not

produce ASTs that are suitable for manipulation. Only a common parsing infrastructure can produce AST structures that allow a rewrite engine and code generating infrastructure to function over arbitrary domain languages and combinations of languages.

- *Customers can become transformation tool builders*. There is significant learning curve in building transformation-based tools. A customer seeking a single tool can save money by letting transformation specialists build it. But transformation methods are well-suited to a range of software life cycle tasks, and customers can be trained to build tools themselves and incorporate the technology into their operation with great benefit and cost savings.

4. FUTURE DIRECTIONS

The PRiSm or CORBA component technologies impose computational overhead as service requests are routed through several new layers of component communication protocol. A DMS-based approach to partial evaluation could relieve this overhead. Essentially, the extra layers exist to provide separation of concern in design and coding and to provide plug-and-play capability at configuration time. With semantic awareness of the component wiring present in the source code, though, a DMS tool could be developed to statically evaluate the various communication indirections, thus sparing that run-time overhead. In this highly performance-sensitive environment, the effort could be well justified.

5. ACKNOWLEDGMENTS

We give our thanks to our collaborator in this effort, the Boeing Company, and to the DARPA PCES program for its funding.

6. REFERENCES

[1] *PARLANSE Reference Manual,* Semantic Designs, Inc. 1998.

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

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

[4] Sharp, D. C., Reducing Avionics Software Cost Through Component Based Product Line Development, *Proceedings of the 1998 Software Technology.*

[5] Gidding, V., Beckwith, B., Real-time CORBA Tutorial, OMG's Workshop on Distributed Object Computing For Real-Time and Embedded Systems, *www.omg.org/news/meetings/workshops/rt_embedded2003.html*, 2003.