

Re-engineering C++ Component Models Via Automatic Program Transformation

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

Ira D. Baxter, Ph.D.
idbaxter@semdesigns.com
Semantic Designs Inc.

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

Brian J. Ellis Kenn R. Luecke
The Boeing Company

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 difficult and expensive. But constructing highly scalable transformation tools supporting modern languages in full generality is itself a painstaking and expensive process. This cost can be managed by developing a common transformation system infrastructure reusable by derived tools that each address specific tasks, thus leveraging the infrastructure costs. This paper describes the Design Maintenance System (DMS¹), a practical, commercial program analysis and transformation system, and discusses how it was employed to construct a custom modernization tool being applied to a large C++ industrial avionics system. The tool transforms components developed in a 1990's-era component style to a more modern CORBA-like component framework, preserving functionality.

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

1. Introduction

DMS is a mature infrastructure for doing analysis and transformation of software based on deep semantic understanding of programs. It supports virtually all conventional software languages and can be applied to systems built from multiple coding and design languages. The Boeing Migration Tool (BMT), built using the DMS infrastructure, automatically transforms the component framework of a large C++ avionics system from a 1990's era model to one based

on a proprietary variant of the Common Object Resource Broker Architecture (CORBA), preserving functionality but introducing regular interfaces for inter-component communication.

We describe the DMS infrastructure and the BMT application itself to provide insight into how transformation technology can address software analysis and evolution problems where scale, complexity, and custom needs are barriers. We illustrate 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. We present Boeing's assessment of the project, assess the return on investment of the automated migration strategy, and present some reflections on the experience to guide others considering large scale code re-engineering projects.

2. The DMS Software Reengineering 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 can accept and simultaneously utilize definitions of 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

¹ DMS is a registered trademark of Semantic Designs Inc.

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.
- 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 name and type analysis and defining and updating arbitrary namespaces containing name, type, and location information with arbitrary scoping rules.
- Attribute evaluation for encoding arbitrary analyses over ASTs with rules tied to grammar elements.
- An AST-to-AST rewriting engine that understands algebraic properties (e.g., associativity, 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 in Figure 1. 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 of which is support for computational scale. DMS is implemented in a parallel programming language, PARLANSE [13], which enables 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,

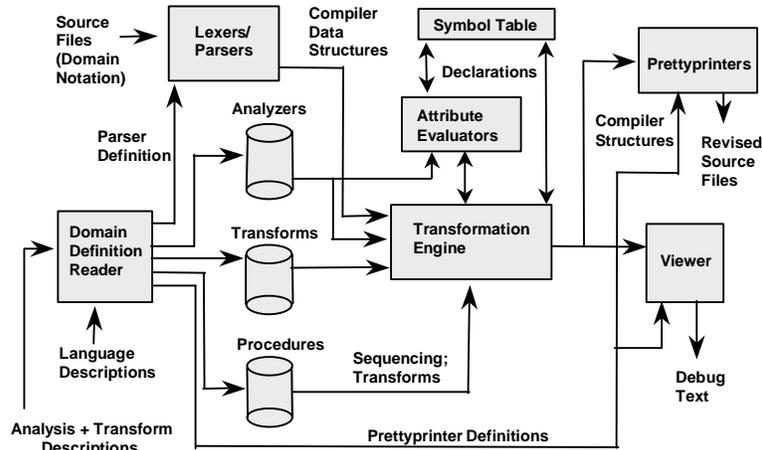


Figure 1 The DMS architecture

transformed, and printed with preprocessor directives preserved, even containing preprocessor conditionals. The C++ name and type resolver has been extended to fully support preprocessor conditionals, creating a symbol table with conditional entries for symbols and conditional relationships between lexical scopes containing such symbols.

DMS has been under development for nine years. As presently constituted, it 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/clone code elimination, test code coverage, execution profiling, source code browsing, and static metrics analysis.

A more complete overview of DMS is presented in [5], including discussion of how DMS was extensively used to create itself. For example, the DMS lexer generator, prettyprinter generator, and its name and type resolution analyzers for various languages are all tools created with DMS. Various other DMS-based tools are described on the Semantic Designs Inc. (SD) web site [16].

3. The Boeing Migration Tool

Boeing's Bold Stroke avionics component software architecture is based on the best practices of the mid-1990's [17]. Component technology has since matured, and the Common Object Resource Broker Architecture (CORBA) component model has emerged as a standard. The U.S. Government's Defense Advanced Research Projects Agency's Program Composition for Embedded Systems (DARPA-PCES) program and the Object Management Group (OMG) are sponsoring development of a CORBA-inspired standard real time embedded system component model (CCMRT) [8], which offers

standardization, improved interoperability, superior encapsulation, and interfaces for ongoing development of distributed, real time, embedded systems like Bold Stroke. Standardization also provides a base for tools for design and analysis of such systems, and for easier integration of newly developed technologies such as advanced schedulers and telecommunication bandwidth managers.

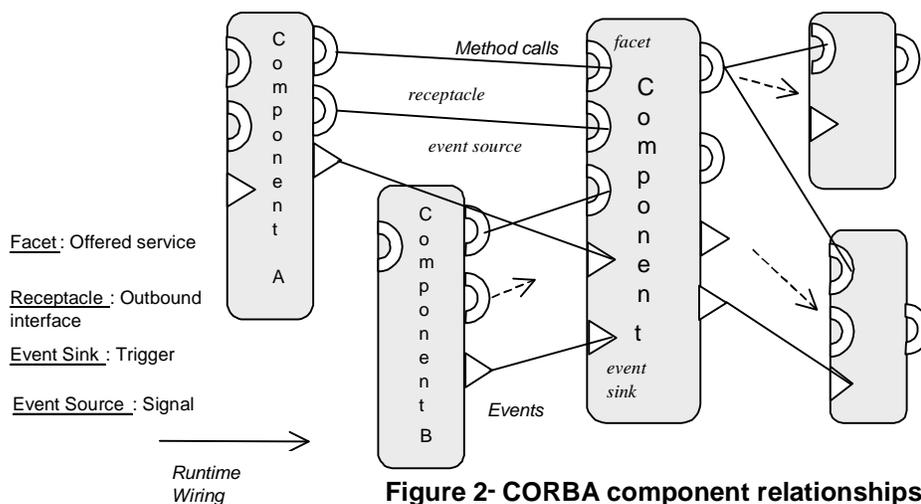
Boeing wishes to upgrade its airframe software to a more modern architecture, a proprietary CCMRT variant known as PRiSm (Product line Real Time embedded System). This will allow more regular interoperability, standardization across flight platforms, and opportunities for integrating emerging technologies that require CORBA-like interfaces. Yet since the legacy software is operating in mature flight environments, maintaining functionality is critical. The modernization effort, then, must not alter functionality as it melds legacy components into the modern component framework.

The task of converting components is straightforward and 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 250,000 lines of C++ source contributes to a typical component, and a sound understanding of a component's name space requires comprehension of all this code.

To deal with the scale, semantic sensitivity, and regularity issues, DARPA, Boeing, and SD decided to automate the component migration using a custom DMS-based tool. DMS, with its C++ front end complete with name and type resolution, its unique C++ preprocessor, its transformation capability, and its scalability, was a uniquely qualified substrate for constructing a migration tool that blended code synthesis with code reorganization. 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, potentially idiosyncratic examples. It also ensures a uniform treatment in a variety of ways including name conventions for new entities, commenting conventions, code layout, and file organization.

Figure 2 diagrams the relationship of communicating CORBA components. Facets are classes that provide related methods implementing a service, i.e., an area of functional concern. Event sinks provide entry points for signaling changes of state upstream, to which the component may wish to react. A similar interface exists for outgoing calls, with event sources being a standard route through which the demand for service is signaled to other components, and receptacles providing connections to the instances of facets for other components. Components are wired together at configuration time, prior to system execution.



While event sinks and facets are very similar ideas with respect to data flow, they are distinct in the CORBA model for several reasons. The functionality of a facet is specific to each facet and component and to the services it offers, and facets share little commonality in form or function with each other or with facets of other components. Event sinks, on the other hand, implement a standard protocol for inter-component event signaling. Though the code specifics vary with the components' functional interfaces, the function, style, and structure of event sinks are consistent across all components, and hence they are given distinct, stylized identity and treatment, likewise for event sources and receptacles.

The legacy component structure is essentially flat, with all a component's methods typically 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, each forming a distinct class populated with methods

reflecting a particular area of concern. Other classes, like the event sinks and receptacles, must be synthesized at a more fine-grained level, extracting code fragments or connection information from the legacy system.

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, SD 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 coding task is formulating facet specifications for all the components to be migrated, a very easy task for a knowledgeable engineer. The facet language itself is defined as a DMS domain, allowing DMS to automatically generate a parser from its grammar and to define specification processing tools as attribute evaluators over the facet grammar. Figure 3 shows a facet specification for a simple, two-facet component. The list of legacy classes prescribe which classes are to be transformed as part of the component. Façade declarations specify the legacy classes in which the facet interface methods appear. Facet method parameters need be supplied only to select from overloaded methods.

Figure 4 illustrates the essential factoring of the monolithic legacy class into CORBA/PriSm facets (but does not reflect the activity related to event sinks, receptacles, or other new PriSm classes.) The methods identified in the engineer's facet specification are relocated into new facet classes. References from within those methods to outside entities are modified with additional pointers to resolve to the originally intended entities. Furthermore, references in the rest of the system to the relocated methods are also adjusted via an indirection to point into the new facet classes. Declarations of all these new pointers must appear in the appropriate contexts, along with #include directives to provide the necessary namespace elements.

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

The BMT's migration process begins with the parsing of all the facet specifications relating to the component. A DMS-based attribute evaluator over the facet domain traverses the facet specifications' abstract

```

COMPONENT AV_LogicalPosition_StateDevice
  FACET AV_LogicalPosition_InternalStatus
  FACET AV_LogicalPosition_StateMode
  LEGACYCLASSES
    AV_LogicalPosition_StateDevice
  END COMPONENT

FACET AV_LogicalPosition_InternalStatus
  FACADE AV_LogicalPosition_StateDevice
    "IsInitDataRequested"
    "IsAlmanacRequested"
    "IsDailyKeyInUseVerified"
    "IsDailyKeyInUseIncorrect"
    "IsGUV_User"
    "IsReceiverContainKeys"
    "GetMissionDuration"
    "IsRPU_Failed"
    "GetMemoryBatteryLow"
    "GetReceiverLRU_Fail"
  END FACET

FACET AV_LogicalPosition_StateMode
  FACADE AV_LogicalPosition_StateDevice
    "bool GetBIT_Status ()"
    "GetPosition_StateRequested"
    "GetPosition_StateAchieved"
    "GetEventSupplierReference"
  END FACET

```

Figure 3 - Facet specification example

syntax trees and collates it into 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 referenced via #include. These would naturally include the files for the façade classes of the neighboring components. The DMS C++ name and type resolver constructs a symbol table for the entire source base, allowing lookup of identifiers and methods with respect to any lexical scope. Only by internalizing the entire code base in this manner can symbol lookups and the transformations depending on them be guaranteed sound. This is one key point that defeats scripting languages for writing C++ transformers.

Four particular kinds of 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. The BMT generates a base class for each facet, essentially a standard form and a "wrapper" class inheriting from the facet and containing one method for each method in the functional facet's interface (i.e., for each method listed in the facet specification). These wrapper methods simply relay calls to the corresponding method in the component's legacy classes. Constructing the

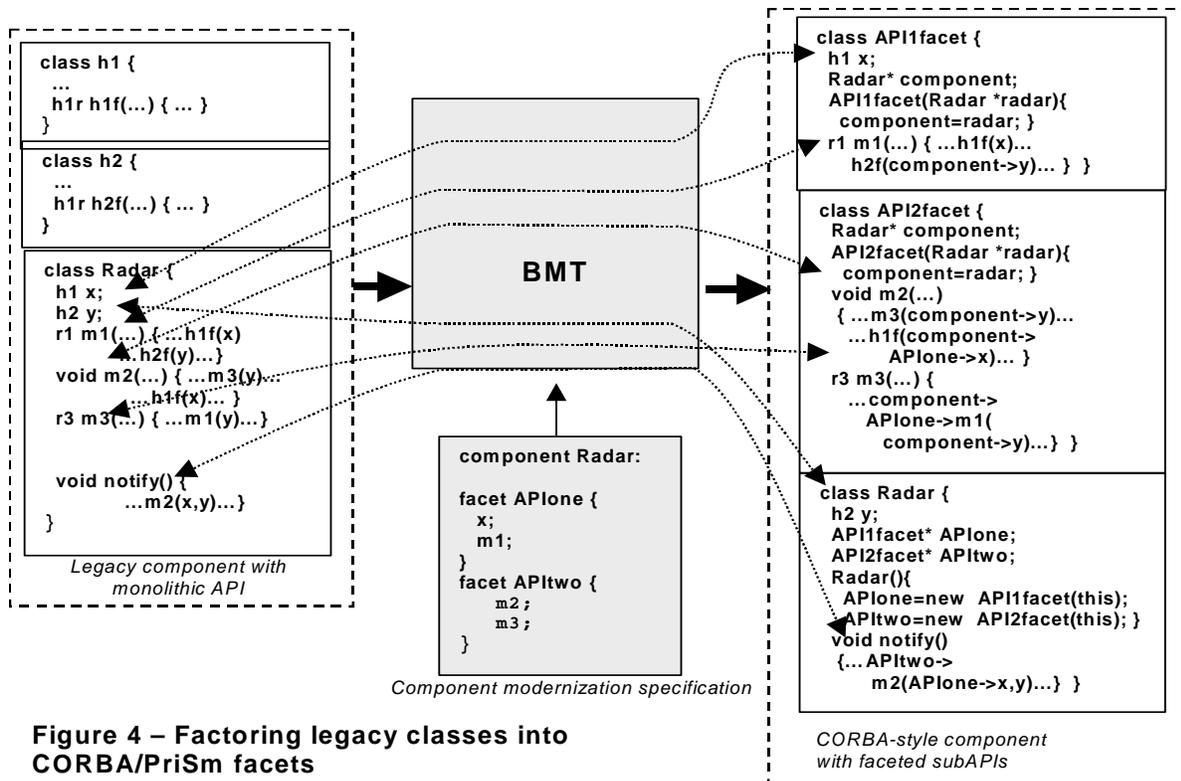


Figure 4 – Factoring legacy classes into CORBA/PriSm facets

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 to access standard component infrastructure. A nest of constructor patterns expressed in the DMS pattern language pull the pieces together into a class definition, thus synthesizing the code from patterns. The change from a pure CORBA scheme like that illustrated in Figure 4 to a wrapper scheme was a major mid-project design change that allowed modernized components to interoperate with untranslated legacy components during the long-term modernization transition.

- After constructing the facets and wrappers, the BMT transforms all the legacy code calls to any of the facets' methods, redirecting original method calls on the legacy class to instead call the appropriate wrapper method via newly declared pointers. The pointer declarations, their initializations, and their employment in access paths are all inserted using source-to-source transforms, the latter with conditionals to focus their applicability. An example of one such transform appears in Figure 5. The various arguments are typed by their corresponding C++ grammar nonterminal names, and the rule transforms one postfix_expression to another. Within the body of the rule, argument names are preceded by "\". The rule uses concatenation to construct the symbol for

the new pointer and adds the indirection. It will be applied only when the "pointer_refers" predicate establishes that the class_pointer and method name correspond to a specified facet method. If the rule does fire, it will also trigger some tool-internal bookkeeping as a side effect.

```

+ Rewrite legacy cross-component method references,
+ e.g., Comp1->methname(arg1, arg2) rewrites to
+ Comp1BarFacetPtr->methname(arg1, arg2)
private rule adjust_component_access(
  class_ptr: identifier, method: identifier,
  arguments: expression_list):
  postfix_expression->postfix_expression
= "\class_pointer->\method(\arguments)"
->
"\concat(\concat(
  \get_facet_name(\class_pointer, \method),
  Facet_), \class_ptr)->\method(\arguments)"
with side-effect remove_identifier_table_entries
(class_pointer, method)
if pointer_to_foreign_component(class_ptr).

```

Figure 5 - DMS access path rewrite rule

DMS patterns and rewrite rules are parameterized by typed AST's from the domain (C++). Quoted forms within the definitions are in the domain syntax, but within quotes, backslashes can precede non-domain lexemes. For example the references to the class_pointer, method, and arguments parameters are preceded with slashes, as are references to the names of other patterns (concat and get_facet_name), and syntactic artifacts of other pattern parameter lists (parentheses and commas).

The application of the rewrite rule can be made subject to a Boolean condition, as it is here by the external function `pointer_refers_class_of_foreign_component`, and may also, upon application, trigger a side effect like external procedure `remove_identifier_table_entries`.

- “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 method calls, identifying which facet the called method belongs to, and generating code to serve each connection accordingly. Figure 6 illustrates a portion of a receptacle .cpp file. The figure illustrates one of the Connect methods, which are always of the same form, but which draw specifics from the calling environment and the specifics of the legacy method. The BMT creates include directives appropriate to the calling and called components and as required by other symbols appearing in the method headers. Standard receptacle comments are inserted.
- Event sinks are classes that are among the communication aspects of a component, representing an entry point through which an event service can deliver its product. They are uniform in style and structure for all components in a way that is more or less independent of the components' functionality, but their content is nevertheless driven by the legacy code's particular interconnectivity behavior. Since the essential code fragments for event processing already exist in the legacy classes (though their locations are not specified to the BMT and cannot generally be characterized), synthesizing event sinks involves having the BMT identify idiomatic legacy event-handling code throughout the legacy component by matching against DMS patterns for those idioms. Code thus identified is moved into the

new event sink class, which is synthesized with a standard framework of constructive patterns. Control structures are then merged to consolidate the handling of each specific kind of event. One necessary complication of this extensive movement of code from one lexical environment to another is that all relevant name space declarations must be constructed in the new event sink. Definitions, declarations, and `#include` directives supporting the moved code must be constructed in the event sink class, and newly irrelevant declarations must be removed from the original environment. Doing all this requires extensive use of the DMS symbol table for the application, which among other things retains knowledge of the locations within files of the declarations and definitions of all symbols. Event sink code extraction and synthesis combines pattern-based recognition of idiomatic forms with namespace and code reorganization and simplification via semantically informed transformation.

The BMT, then, significantly modifies the legacy classes for the component by extracting code segments, by modifying access paths, by removing component pointer declarations and initializations, by adding facet pointer declarations and initializations, by reconfiguring the namespace as necessary, and by doing other miscellaneous modifications. It also introduces new classes for the component's facets, facet wrappers, receptacles, event sinks, and an

```
// File incorporates PRiSm Component Model (Wrapper version)
// file generated by the BMT tool for the PCES II Program
#include "AMV__LogicalPosition_StateDevice/AMV__LogicalPosition_StateDevice.h"
#include "AMV__LogicalPosition_StateDevice/AMC__Position1Receptacle.h"
#include "AMV__LogicalPosition_StateDevice/AMC__Position1Wrapper.h"
AMC__Position1Receptacle::AMC__Position1Receptacle(
    AMV__LogicalPosition_StateDevice * theAMV__LogicalPosition_StateDevicePtr)
: theAMV__LogicalPosition_StateDevicePtr_(theAMV__LogicalPosition_StateDevicePtr)
{
    //There is nothing to instantiate.
}
AMC__Position1Receptacle::~AMC__Position1Receptacle()
{
    //Nothing needs to be destructed.
}
bool AMC__Position1Receptacle::ConnectAMV__LogicalPosition_StateDevice (
    BM__Facet *item)
{
    // Cast the parameter from a BM__Facet pointer to a wrapper pointer
    if (AMC__Position1Wrapper *tempCompPtr =
        platform_dependent_do_dynamic_cast<AMC__Position1Wrapper *>(item))
    {
        theAMV__LogicalPosition_StateDevicePtr_ ->
            AddAMC__Position1Connection(tempCompPtr);
        return true;
    }
    ...
}
```

Figure 6 - A portion of a generated receptacle.cpp file

“equivalent interface” class desired by Boeing component designers. Each kind of new class has a regular structure, but the details vary widely, based on the characteristics of the component being translated. Some classes, like the event sinks, are populated with code fragments extracted from the legacy class and assembled into new methods under some collation heuristics. In our experience, the amount of new or modified code the BMT produces in a converted component amounts to over half the amount of code in the legacy component. Since a typical component-based system involves a large number of components, this makes a clear economy-of-scale argument for using automatic transformation in component re-engineering.

4. Experience

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. Other operational issues in the conversion were the strict proprietary nature of most of the source code, uncertainty about what exotic C++ features might turn up in the large source code base, Boeing’s evolving understanding of the details of the target configuration, and the geographical separation between Boeing and SD.

To deal with most of these issues, Boeing chose a particular non-proprietary component and performed a hand conversion, thus providing SD with a concrete image of source and target and a benchmark for progress. The hand conversion forced details into Boeing’s consideration. New requirements developed in mid-project, modifying the target (on one occasion very significantly). The flexible DMS approach allowed SD to adjust the tool accordingly and with manageable reworking. Had a manual conversion altered its course at the same point, the cost for re-coding code that had already been ported would have been very high.

Being unburdened by application knowledge, SD was able to focus purely on translation issues, removing from the conversion endeavor the temptation to make application-related adjustments that could add instability. Electronic communication of benchmark results provided a basis for ongoing evaluation, and phone conferences supported development of sufficient bilateral understanding of tool and component technologies, minimizing the need for travel.

SD’s lack of access to the full source code base required the tool builders to prepare for worst case scenarios of what C++ features would be encountered

by the BMT in the larger source base. This forced development of SD’s C++ preprocessing and name resolution infrastructure to handle the cross product of preprocessing conditionals, templates, and macros. These improvements both hardened the tool against unanticipated stress and strengthened the DMS infrastructure for future projects.

5. Evaluation

Boeing used the BMT to convert two large components handling navigation and launch area region computations to the PRiSm Component Model for use in the successful DARPA PCES Flight Demonstration in April, 2005.

Tool input was extremely easy to formulate. One script, which could be shared among all components, set environment variables guiding the tool into the source code base and then invoked the tool. Formulating component-specific input took generally around 20 minutes per component. This time commitment could be less for smaller components and for engineers more experienced using BMT. The BMT converts one component at a time. (A batch file could very easily be written to convert multiple components at once). Usually the tool would execute for approximately ten minutes on a Dell 610 Precision Desktop with dual processor, producing a re-engineered component ready for post-processing by the human engineer. The generated and modified code conformed to Boeing’s style standards and was free of irregularities and errors.

Human engineers did a modest amount of hand-finishing, mostly in situations understood in advance to require human discretion. In these cases, the tool highlighted the generated code with comments recommending some particular action and sometimes generated candidate code. Roughly half the time, this candidate code was sufficient; other cases required hand modification. The most difficult part of the conversion was the event sink mechanization. For input components that had their event mechanization spread across several classes, the BMT correctly moved the code fragments into the event sinks, but the human engineer was required to update some code inside the transitioned code fragments, making judgements, for instance, about whether some aggregations of code should be encapsulated in new functions. These judgement-based decisions would have been necessary with a hand conversion as well. More extensive engineering of the BMT might have eliminated this manual requirement by applying heuristics.

Testing consisted of very close code review and testing of the benchmark component, plus visual inspection and conventional testing of other components.

Among the advantages of using the BMT were:

- All code was generated on the basis of a simple facet specification. The facet specifications were extremely easy and quick to write. The generated code was always complete with respect to the specifications.
- No negative side effects were generated by the tool. This means the tool did not generate code in places that it was not desired, nor did the tool move any code fragments to places that were not desired, mistakes that scripting approaches are more prone to making.
- The tool's name resolution capability proved to be a major advantage over the scripting approaches usually used, which often create unpredictable side effects that BMT is capable of avoiding.

By using the BMT, we reduced the time necessary to convert the two components used for the DARPA PCES Demonstration by approximately half, compared to the time required to convert the components by hand. This time commitment is based on the entire conversion process, which consists of converting the component, post processing the component based on the BMT-generated comments, writing the component's dynamic configuration manager by hand, and testing and integrating the converted component in the resultant software product. For a large project with hundreds or even thousands of components a 50% reduction in total conversion time enables a tremendous cost and time reduction. The tool-based approach would represent the difference between machine-hours and man-centuries of code development labor, between feasibility and infeasibility of mass conversion.

6. Return on Investment

Developing a custom migration tool takes a significant effort. Not including the DMS environment contribution, the BMT required 13,000 lines of new tool code. This development cost must be balanced against the alternative cost of doing the tool's work by hand, so the tradeoff for mechanized migration depends predominately on the amount of code being ported. For small applications, the effort is not worthwhile, but with even a modest sized legacy system, the economics quickly turn positive.

One benchmark legacy component conversion gives an idea of the scale of this conversion. The legacy component, typical in size and complexity, contained 9,931 lines of source code. The BMT-converted component contained 9,456 lines, including 2,109 lines of code in newly generated classes and 2,222 modified lines in the residual legacy classes. Scaling these numbers, to convert a mere 60 components would require revision or creation of over 250,000 lines of code. Porting four such components

would cause more new code to be written than went into the BMT itself. With airframes typically containing thousands of components, the economic advantage of mechanized migration is compelling.

The BMT automates only the coding part of the migration. Testing and integration are also significant factors, and some hand polishing of the BMT output was required. Coding time was reduced to near zero, and the savings in coding time alone allowed a reduction of approximately half the total time required to migrate the components used for the DARPA PCES demonstration. The regularity of style in automatically migrated code provides a less quantifiable but worthwhile extra value.

The measure of economic success is not whether a migration tool achieves 100 percent automation, but whether it saves time and money overall. Boeing felt that converting 75 percent of the code automatically would produce significant cost savings, a good rule of thumb for modest-sized projects. Anything less puts the benefit in a gray area. The code produced by the BMT was 95 percent to 98 percent finished. This number could have been driven higher, but the additional tool development cost did not justify the dwindling payoff in this pilot project.

Cost-benefit tradeoffs should be considered when scoping the task of a migration tool, even while the project is in progress. In this project, for example, we could have developed elaborate heuristics for consolidating event sink code, but we judged the expense to not be worthwhile for the pilot project. Any project of this kind would face similar issues.

Huge projects would easily justify greater refinement of a migration tool, resulting in less need for hand polishing the results, and thus driving coding costs ever lower. Mechanization can mean the difference between feasibility and infeasibility of even a medium size project.

7. Technological Barriers

One inherent technical difficulty is the automatic conversion of semantically informal code comments. Though comments are preserved through the BMT migration, what they say may not be wholly appropriate for the newly modified code. Developing accurate interpretations of free text discussing legacy code and modifying legacy comments to reflect code modifications would challenge the state of the art in both natural language and code understanding. So while new documentation can be generated to accurately reflect the semantics of new code, legacy documentation must be viewed as subject to human revision.

Though the DMS C++ preprocessor capability, with its special treatment of conditionals, was up to the task for this migration, extremely extensive use of

C/C++ preprocessors exploiting dialect differences, conditionals, templates, and macros can lead to an explosion of possible semantic interpretations of system code and a resource problem for a migration tool. Preserving all these interpretations, however, is necessary for soundness. Furthermore, since macro definitions and invocations must be preserved as ASTs through migration, macros that do not map cleanly to native language constructs (e.g., producing only fragments of a syntactic construct or fragments that partially overlap multiple constructs) are very difficult to maintain. Though these unstructured macro definitions cause no problem for compilers, since they are relieved prior to semantic analysis with respect to any single compilation, to preserve them in the abstract representation of a program for all cases is quite difficult.

All these factors suggest that for some projects using languages involving preprocessors, a cleanup of preprocessor code prior to system migration is in order. For reasons of scale and complexity, this is a separate problem that could be tackled with another automated, customized tool.

8. Observations

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 soundness, 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 application owner'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 tool 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 DMS infrastructure. Only 13,000 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 generation infrastructure to function over arbitrary domain languages and combinations of languages.
- *Customers can become transformation tool builders.* There is a 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 engineers can be trained to build tools themselves and incorporate the technology into their operation with great benefit and cost savings.

9. Related Work

Source-to-source program transformations were originally conceived as a method of program generation in the 1970s [2], and the technology has been developing since [10, 11]. The idea that transformations could be used for software maintenance and evolution by changing a specification and re-synthesizing was suggested in the early 80s [4]. Porting software and carrying out changes were suggested and demonstrated in the late 80s [1, 9]. Theory about how to modify programs transformationally using previously captured design information was suggested in 1990[3]. Refine [6,14] was a groundbreaking software transformation engine which was used as a basis for some blackbox commercial automated migration projects. But program transformation as a serious tool for software evolution is largely unrealized in practice.

Mechanical refactoring [13] was proposed in 1990 as a technique for restructuring programs and was recently popularized [7] as a methodology with suggestions for tool support. Tools for refactoring SmallTalk [15] and Java have started to appear, and some experimental work has been done in refactoring C++ [18]. The better Java tools [19,20] do some sophisticated refactorings such as ExtractMethod; others in the market require some manual validation of the steps.

10. Future Directions

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

Semantics-based analysis can also be applied to deeper partial evaluation of the code resulting from the dynamic assembly of components into a flight configuration. For example, code that supports configurations that are not in play and conditionals that dynamically test for those configurations can be eliminated. Indirectly called procedures can be inlined, avoiding indirection and call overhead. Combining automated analysis and code transformation at build time should enhance performance.

11. Acknowledgements

We thank our collaborator in this effort, the Boeing Company, to the DARPA PCES program for funding, and to Lorraine Bier for document preparation.

12. References

- [1] G. Arango, I. Baxter, C. Pidgeon, P. Freeman, "TMM: Software Maintenance by Transformation", *IEEE Software* 3(3), May 1986, pp. 27-39.
- [2] R. M. Balzer, N. M. Goldman, and D. S. Wile, "On the Transformational Implementation Approach to Programming", *Proceeding, 2nd International Conference on Software Engineering*, Oct. 1976, pp. 337-344.
- [3] I. Baxter, Transformational Maintenance by Reuse of Design Histories, Ph.D. Thesis, Information and Computer Science Department, University of California at Irvine, Nov. 1990, TR 90-36.
- [4] I. Baxter, "Design Maintenance Systems", *Communications of the ACM* 35(4), 1992, ACM.
- [5] I. D. Baxter, C. Pidgeon., and M. Mehlich, "DMS: Program Transformations for Practical Scalable Software Evolution". *Proceedings of the 26th International Conference on Software Engineering*, 2004.
- [6] S. Burson, G. B. Kotik, and L. Z. Markosian, "A Program Transformation Approach to Automating Software Reengineering", *Proceedings of the 14th Annual International Computer Software & Applications Conference (COMPSAC 90)*, IEEE Publishers, 1990.
- [7] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison Wesley 1999.
- [8] V. Gidding, B. Beckwith, "Real-time CORBA Tutorial", OMG's Workshop on Distributed Object Computing for Real-Time and Embedded Systems 2003, www.omg.org/news/meetings/workshops/rt_embedded2003
- [9] W. L. Johnson and M. S. Feather, "Using Evolution Transforms to Construct Specifications", M. Lowry and R. McCartney (eds.), Automating Software Design, AAAI Press, 1991.
- [10] E. Kant, F. Daube, E. MacGregor, and J. Wald, "Scientific Programming by Automated Synthesis", in: Michael R. Lowery and Robert D. McCartney (eds.), Automating Software Design, MIT Press, 1991.
- [11] J. Neighbors, "Draco: A Method for Engineering Reusable Software Systems", in: T. Biggerstaff and A. Perlis (eds.), Software Reusability, ACM Press 1989.
- [12] W.F. Opdyke, Refactoring Object-Oriented Frameworks, PhD Thesis, University of Illinois at Urbana-Champaign. Also available as Technical Report UIUCDCS-R-92-1759, Department of Computer Science, University of Illinois at Urbana-Champaign.
- [13] *PARLANSE Reference Manual*, Semantic Designs, 1998.
- [14] Reasoning Systems, Palo Alto, CA, "Refine Language Tools", 1993.
- [15] D. Roberts, J. Brant, R. Johnson and W. Opdyke, "An Automated Refactoring Tool", *Proceedings of ICAST '96: 12th International Conference on Advanced Science and Technology*, Chicago, Illinois. April, 1996.
- [16] Semantic Designs, Inc., www.semanticdesigns.com.
- [17] D. C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development", *Proceedings of the 1998 Software Technology Conference*.
- [18] L. Tokuda and D. Batory, "Evolving Object Oriented Designs with Refactoring", *Proceedings of the Conference on Automated Software Engineering*, IEEE, 1999.
- [19] www.intellij.com. IDEA refactoring tool for Java.
- [20] www.instantiations.com. Jfactor refactoring tool for Java.