The Design Maintenance System® (DMS®)
A Tool for Automating Software Quality Enhancement

Ira D. Baxter, Ph.D.
Semantic Designs, Inc.
www.semdesigns.com

Keywords
software quality, design capture, knowledge reuse, maintenance, automation, testing

Abstract

This white paper sketches how a new type of software engineering technology can be used to automate a variety of software quality enhancement activities. This technology, a kind of extremely configurable, generalized compiler, is packaged in a tool called the "Design Maintenance System" (DMS). Automation is enabled by "teaching" DMS critical concepts of the application problem domain, properties of the application programming language, and methodological software-engineering problem solving approaches. Value comes from using DMS to apply this organization-specific knowledge to frequent activities of its engineers.

In essence, DMS is reusing deep engineering knowledge (as opposed to tools that simply reuse "code"). This amplifies the effect of skilled engineers, by allowing them to focus on the deep engineering issues rather than the microscopic details of carrying out such engineering tasks, and by allowing them to experiment more easily with changes that are system wide.

DMS can be used for a variety of software quality enhancement activities. Because DMS can be configured with many kinds of knowledge, there are many possibilities. A few examples:

- *Finding/removing dead or redundant code*
- *Instrumenting code for test coverage*
- *Test generation from specifications*
- *Checking organizational coding standards*
- *Reshaping code to regularize structure (error handling, etc.)*
- *Extracting documentation from source code*
- *Reading/checking high-level specifications (comunication..protocols,tests, state machines)*
- *Code generation from high level specifications*
- *Porting code to new dialects/environments*

In a time of tight staff, budget and development cycles, growing software systems, and rising expectations of product quality, DMS can deliver considerable competitive value to the employing organization.

1 Automation and design knowledge

Classic software engineering employs vast amounts of manual engineering and only small amounts of automation. Productivity gains are most likely to come from enhanced automation harnessing reusable *design* knowledge, such as the kinds of problems encountered in a problem domain, different methods and tradeoffs for solving those problems, means for coding solutions, test methods, etc. ("Code reuse" is a special kind of low level kind of design reuse).

Conventional compilers provide most of the present-day automation leverage, by analyzing low-level programs (C, Java, Cobol) for mere language legality, and translating those programs to even lower levels (machine code). Analysis and translation are keys to more automation, but not at the low level of conventional compilers.

If only the source code is analyzed or modified, design documents start to decay and soon become untrustworthy or useless. Useless design documents explains why most software engineers spend half their time examining code trying to understand what it does and why. Technically, this is a hopeless task because product requirements and specifications are not in the source code!

It is well known that errors made early in the design process, but discovered late, are much more difficult to fix than errors made and found late in the implementation process [Boehm81]. A specification error costs 100 times as much to fix as a coding mistake! This suggests that tools that aid the correct construction of designs, and tools for analyzing designs to detect errors, would be extremely helpful. But since engineers and organizations are fallible, it also implies that tools that can track such design information into target systems and make corresponding changes will also be valuable, to minimize the costs of fixing such inevitable errors.

Automation can only help when it "understands" what is being automated; this means that such automated help *must* understand designs. So to maximize leverage, designers and their tools must work on *designs*, not just directly on the code.

The implications are that tools for design automation must eventually handle design information (specifications, architectures, and the decisions accepted and rejected that

lead to the actual software and hardware implementation). The ultimate goal is to enable the construction and maintenance of a continuously up-to-date, modifiable design document and corresponding system.

Design management tools must:

- Explicitly capture and maintain the design knowledge used in a system
- Manage the scale of the design information involved (thousands of concepts and their relationships) and the software system source files (millions of lines),
- Provide long term value in managing designs,
- Provide *near term value* via analyses, and engineer-directed change management of existing software systems for which design information is difficult to obtain.

Key design knowledge to be captured and reused consists of:

- *Domain Notations:* how to specify a problem in a particular problem domain or engineering notation. (obtained from domain analysis [Neighbors84])
- *Specifications:* the concise formal description of problem, stated in a particular domain notation
- *Generative Knowledge:* knowledge of *possible* ways to implement problem solutions using target domains
- *Implementation Knowledge:* how a particular system is structured to carry out its specification.

Building tools to manage design information is extremely difficult, and cannot be done overnight. First, there must be theory about what designs are, and how those designs can be captured, analyzed and modified, and then tools can be constructed used the theory foundations.

Semantic Designs (SD) is committed to constructing such design management tools and delivering their value to engineering organizations. The technical founders of SD have worked out a unique theory for design management over the past 15 years, based on a generalization of compilers called *transformation systems* [Baxter92]. SD has spent the last 5 years constructing a unique toolset moving toward this the vision called the "*Design Maintenance System (DMS)*.

The DMS toolset, over time, provides increasing ability to capture, analyze and revise the design and source code of large software systems. Such full-lifecycle automation would provide enormous productivity leverage to the organizations' engineering staff, and thus to the quality and market value of the organization's products.

While the present version of DMS [Baxter2004] does not realize all the benefits of complete design management, it can deliver considerable value by automating many tasks related to source code and specifications. This paper concentrates on the near term capability of DMS to analyze and modify large software systems without having large amounts of available design knowledge.

2 What DMS does

The current DMS Reengineering Toolkit (http://www.semanticdesigns/Products/DMS/DMSToolkit.html) enables an organization to define and automate:

- *complex analysis tasks over a large software base* (dead code detection, style checking, testing, bug detection, documentation extraction)
- *massive regular change tasks to software* (test coverage probe insertion, API changes, reformatting, structural changes)
- *source code generation from a specification* (program generation, test case generation)
- *translations from one notational system to another* (porting legacy applications, language upgrades)

3 DMS overview

DMS is able to accomplish this huge variety of tasks by using the theory of design information to unify many seemingly disparate activities with a common foundation: that of a *configurable, generalized compiler, parameterized by considerable general and domain-specific knowledge*. The process requires two stages: first, encoding knowledge for DMS to harness, and then reaping value by applying the encoded knowledge to the analysis or modification of software sources (see Figure 1). The knowledge given to DMS is essentially the knowledge used by engineers:

- *Language definitions*, i.e. what notational systems are involved in the system and/or analyses of the system (the C language, state machines, etc.)
- *General analysis methods* for each type of language (bad pointer analysis, what state transitions can occur, etc.)
- *General Optimization and Implementation methods* (how to optimize Boolean equations, how to implement queues, etc.)
- *Specific methods for analyzing* the type of problems faced by the engineer's organization (Are the right communications procedures called? Is this state of the program ever reached?)
- *Specific methods for making changes* relevant to the organization (optimizing large buffer transfers, implementing using RTOS primitives)

As an example task, the figure shows an "example" large-scale software source being optimized by DMS using supplied background knowledge. The language definitions are used to parse the large software system. The change-specific analyses are used to extract the easily found fact, "*a in b*", from those sources. The background knowledge about this system indicates the fact "*b in c*" is true. More

change specific analyses concludes "*a in c*" from the previous two facts, and tells the engineer in an analysis result. Finally, the "*a in c*" conclusion is used to automatically optimize the program using the general transforms, and produce a revised result, using the language definitions to pretty-print the result. The result: automated optimization of the software system.

DMS provides economic value because the effort to encode this knowledge is much smaller than the effort to manually carry out tasks using this knowledge, especially if the task is frequently repeated or the volume of code is large. Typically, DMS is configured by providing it with a few thousands of lines of "definitions". This is generally very small compared to the hundreds of thousands of lines of code processed by DMS in carrying out designated tasks.

Semantic Designs enhances this economy of scale by providing off-the-shelf modules for commonly used language definitions, general analyses and transforms, such as for C, COBOL, and XML etc. So the organization using DMS may need only to supply their task-specific knowledge. SD also offers services in defining tasks appropriate for DMS, and encoding the application-specification knowledge.

An additional valuable benefit of this arrangement is that the organization defines and captures its key engineering knowledge in a form usable by later engineers.

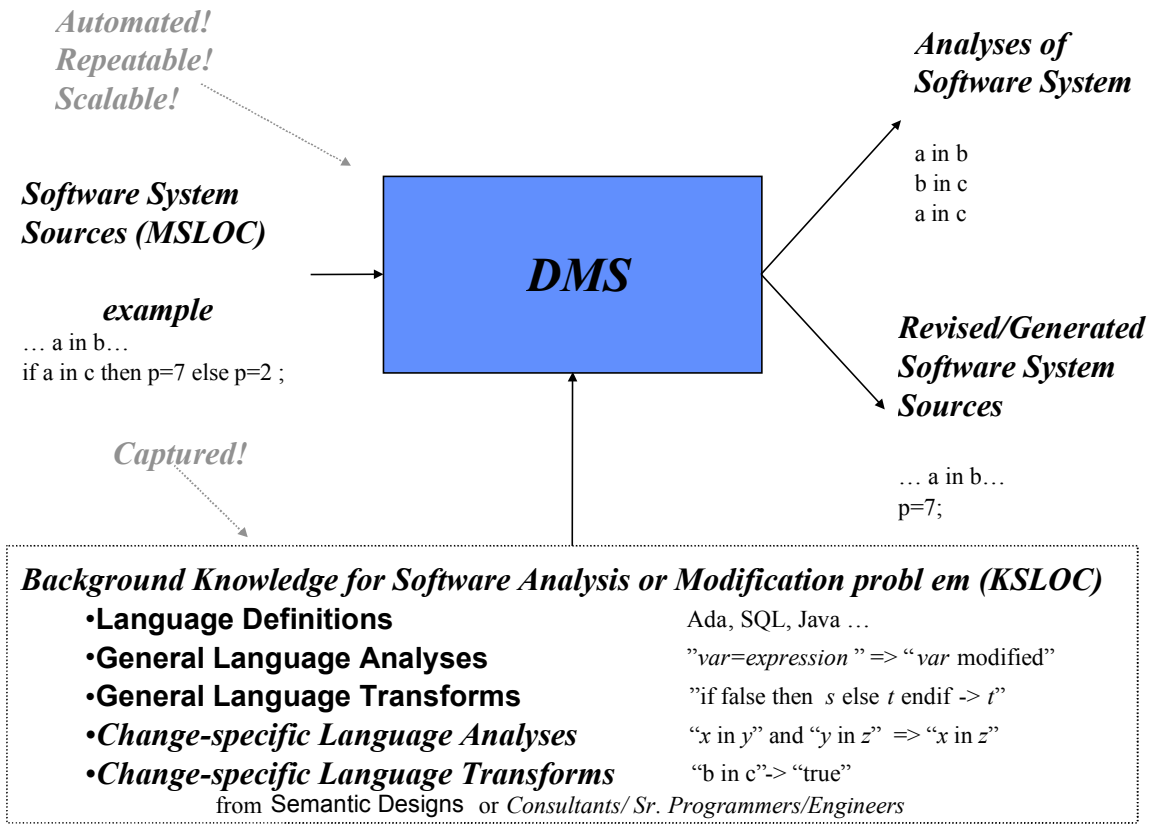

Figure 1: Automating a Software Engineering Task

4 When to use DMS

The key to extracting value with DMS is to think of software engineering tasks in terms of the activities defined in section 2. As an example, one might consider determining whether all parts of an application program have been executed as an analysis task. Removing dead code would be a massive regular change. Producing code to implement a protocol from a specification of the protocol transitions is source code generation. And converting from COBOL74 to COBOL85 is a translation. Once a task is cast in these terms, the DMS mechanisms offer obvious solutions to various types of tasks.

The first requirement is that the engineering problem have enough scale to justify encoding the appropriate knowledge. The basic issue is, how long will it take to configure DMS versus how long will it take the engineers by traditional methods? There are many simple one-shot tasks for which the payoff of encoding this knowledge is

not worth it. These tasks should continue to be done by manual methods; typically a man-day task is too small to handle with DMS. However, simple tasks that are frequently repeated, such as extracting documentation, can often justify using DMS. Tasks that require months of man-effort (test construction, code reorganization, etc.) clearly warrant evaluation with respect to DMS capabilities.

For many common tasks (e.g., "Draw a call tree", "Determine test coverage", "Reformat the sources") in standard languages (e.g., Java), an organization can usually find a vendor supplying a tool for that specific task. For such standard tasks, DMS provides no particular advantage (although using DMS for such tasks might make sense if DMS served other purposes in the organization too, just to minimize tool learning costs).

However, there are many high-value nonstandard tasks, limited usually only by the cleverness of the engineers ("find duplicated code", etc.) for which tools are not available from vendors. Similarly, there are many standard tasks on nonstandard languages ("Z8000 C") or environments (e.g., embedded systems). DMS is often useful in these contexts as being the *only* practical means to obtain a tool.

5 Sample DMS Applications and Benefits

The DMS tools have been applied in practice to many code analysis and modification tasks. We list a few below:

1) Itself. (Yes, we use our own tools!) DMS provides many small languages to specify its parts, and generates a considerable portion of it automatically. An example: DMS automatically converts a 30K SLOC COBOL analyzer spec to roughly a *million* SLOC of code.

2) Factory code generation. A code generator was built that converts a high-level factory machining process specification into assembly level controller code for a factory automation computer.

3) Large-scale equation simplification: A 40K term Boolean equation was simplified by 80%.

4) Code clone detection and removal. DMS has been used to find 12% redundancy in 800K SLOC COBOL, and 14% redundancy in 2.5 million lines of Java, and 9% in 400K SLOC C code [BaxterEtAl98].

5) Elimination of dead preprocessor conditionals from 1.5 million lines of source code across 1800 C files.

The economic benefits of these applications depends on the organizations' view of value. As one example, for DMS self-application, the payoff is that DMS otherwise simply would have been impossible to construct! The factory automation code generation also fits in the otherwise almost impossible to construct category.

For clone detection, the payoff is in terms of reduced engineering costs. It costs roughly (1999 terms) $US1.00/source-line/year to have running code in an organization. Removing the 10% redundant code from a million line system then saves an organization $US100K every year until the end of the software life.

For one customer, deletion of dead preprocessor conditionals was necessary to manage the code base. Manual estimates for the job suggested a man-year of effort. DMS accomplished the job in 2 days, cutting 2.5 months out of the development schedule.

6 Why not build tools from scratch?

It is easy to find economic incentive in organizations to build source analysis or modification tools. Consequently it is often attempted, usually based on generally respected tools such as PERL, LEX and YACC. This generally fails, as the organization discovers:

- That PERL, which extremely good at pure text strings, has no real understanding of source program structures, is therefore unreliable in its ability to detect and chance source code, and can therefore only be used for 90% solutions of very simple tasks. The remaining 10% must be done by hand, and this is impractical on million line systems.
- Defining working parsers for real languages is hard, because modern languages have very large and complex rules (SD's definition of COBOL is 3500 syntax rules alone!)
- LEX and YACC are far less help for parsing than their reputation suggests (e.g., most languages are not "LALR(1)" as YACC requires, so the language definition must be twisted to make YACC work with it; a practical tool must report errors, and YACC has no real error reporting ability)
- Often, to process a language such as C, one has to support some kind of preprocessor handling possibly nested include files, conditional selection, macro definition and expansion, for which YACC provides zero support
- A working parser barely scratches the surface of building a complete tool. One must have the ability to analyze the parsed result, modify the parsed result, and regenerate source text

compatible with conventional compilers and tools from the modified result.

- Scalability must be addressed if the tool is to work in practice. A useful tool must be able to parse hundreds of files, possibly millions of lines of code as part of a single analysis in a single session. It must handle the multiple languages used in an application (e.g., C, SQL, XML, etc.). And it must have the computational muscle to process this large amount of input in a reasonable length of time.

As a consequence, such in-house tools generally consume the full-time attention of highly-skilled engineering resources, but are usually are not completed, cannot be fielded to user engineers, or maintained by anyone other than the principal author. Many organizations either give up or never tackle automation in software engineering, even if the economics of a working tool make enormous sense.

Semantic Designs has designed DMS to provide an integrated tool infrastructure, so that engineers can concentrate on the details of the task at hand. This makes it economically practical for an organization to construct, field and maintain automated scalable, nonstandard software engineering tasks using DMS as a foundation.

7 How DMS does it: A Generalized Compiler

Conventional compilers contain several standard structures:

A. *A language parser*, designed specifically to read the language for which the compiler was designed, reports on incorrect syntax, and builds internal compiler data structures representing the program

B. *A symbol table*, used to keep track of names and types of language-specific entities such as variable and functions, and handling language scoping rules

C. *One of more analysis components*, to check the semantic integrity of the program ("type checking") and to detect usage patterns and information flows in code to enable optimizations. The analysis components may produce readable results (e.g. "unassigned variables" or "cross-reference")

D. *A code transformer/optimizer*, which uses the program representation data structures and analysis results to reshape the program into a more efficient one (e.g., doing compile-time arithmetic on constants, in-lining subroutines, moving code out loops)

E. *A code generation component*, which uses the program representation data structures and the results of the analyzers to choose appropriate target machine idioms

DMS contains generalizations of these compiler components for many reasons:

- It needs to be configurable in all "compilation" aspects to carry out arbitrary analyses and transformations
- It is designed to work in an arbitrary language, rather than just one.
- It must work with several languages at once, not just one at a time
- It must work with many files at once, not just one at a time
- It needs to be used for forward and reverse engineering, not just compilation

Consequently, DMS provides the following generalized compiler components:

a) *An arbitrary language parser*. This component reads a set of designated language(s) of interest and builds compiler-like data structures that can be interpreted by the rest of DMS. It converts language tokens such as string and floating point numbers to the computer's native representation, captures comments, formats of values (radix, leading zero count) and source positions (file, line number and column numbers). It reports incorrect syntax, and automatically builds internal DMS data structures called Abstract Syntax Trees (ASTs), efficiently and compactly representing the source files and their structure. Infrastructure to support INCLUDE files and preprocessors is built in (and a complete C/C++ preprocessor is available).

b) *A generalized symbol table*, that can store names and information related to names, with arbitrary language scoping rules.

c) *A generalized parallel-execution analysis engine*, called an attribute evaluator. This component is used to define semantic checks over the ASTs for the designated languages, and to define detectors of usage patterns and information flows in the source files The analysis components may be configured to produce readable results (e.g. "dead code in File X at Line Y") or compute summary results for use by other DMS components.

d) *A general transformation ("rewrite") engine*. The engine uses rules and patterns specified directly using the syntax of the designated languages (i.e., in the engineer's vocabulary), to

find places in the ASTs where transforms should occur and make changes to those places, possibly contingent on analysis results (e.g., "match variable declarations; if no use of the variable anywhere, delete the variable declaration"). The transformation engine can directly carry out an astonishing variety of effects. This includes classic compiler optimizations (folding constants, in-lining subroutines, moving code out loops).

e) *A generalized code generation component*, which is just the rewrite engine used across languages and/or abstraction levels. It can perform high-level code to low-level code translation (e.g., mapping "while loops" to "conditional branches and jumps", or "XML DTDs" to "Java structures". And it can perform same-level to same-level transformations (e.g., translating "Visual Basic" to "C").

f) *A prettyprinter,* which can regenerate the appropriate language text files from the original and more importantly, from the modified ASTs. This provides formatted, readable regenerated source files compatible with other language processing tools such as conventional compilers.

g) *Domain Language definition tools*, which enable DMS users to define their own design or implementation languages (or dialects), attribute evaluator equations, or rewrite rules. (Semantic Designs can supply suitable DMS-tested language definitions for a large number of standard languages, such as C, C++, Java, COBOL, XML, SQL, PL/I, Fortran, Verilog, VHDL…)

h) *Scalable infrastructure in space and time*. DMS is unique in terms of scale management, as required by the amount of information used in today's large software systems. DMS can read tens of thousands of source files, totaling to several million lines of code in multiple languages into a single DMS session on a commodity Intel workstation, keeping track of their origin in case error reports or updates are required. Because of the amount of information, much of DMS is designed to execute in parallel on Windows NT systems. As an example, DMS may have to apply such analyses to thousands of files, so the attribute evaluator automatically executes in parallel where possible.

Summary

DMS is a revolutionary software engineering tool, designed to aid engineers by automatically carrying out analyses and modifications of software systems. These are cast in a way that allows DMS to be applied to a large number of useful engineering activities, such as checking, testing, code generation, translation and many others. DMS's strength comes from foundations rooted deeply in a theory of Design Maintenance, and its implementation designed to handle large scale systems composed of millions of lines of code and tens of thousands of files.

DMS will reduce engineering time and raise product quality by capturing and reusing knowledge that is core to the engineering organization. It will be a key tool of every engineering organization, just as editors and compilers are key tools today.

References

[Baxter92] I. Baxter. 1992. *Design Maintenance Systems,* Comm. of the ACM 35(4), Apr 1992, ACM.

[BaxMeh97] I. Baxter and M. Mehlich. *Reverse Engineering is Reverse Forward Engineering*. 4th Working Conference on Reverse Engineering, 1997, IEEE

[BaxPidg97] I. Baxter and C. Pidgeon. *Software Change Through Design Maintenance.* International Conference on Software Maintenance, 1997, IEEE Press.

[BaxterEtAl98] I. Baxter, et. al *Clone Detection Using Abstract Syntax Trees*, International Conference on Software Maintenance, 1998, IEEE.

[Baxter2004] I. Baxter, C. Pidgeon, M. Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution, in Proceedings of the International Conference on Software Engineering, 2004, IEEE Press

[Boehm81] B. Boehm, Software Engineering Economics, Prentice-Hall, 1981

[Neighbors84] J. Neighbors. *The Draco Approach to Constructing Software from Components*. IEEE Transactions on Software Engineering 10(5):564-574, 1984.

