DMS®: Program Transformations for Practical Scalable Software Evolution

Ira D. Baxter Christopher Pidgeon Michael Mehlich
Semantic Designs, Inc.
{idbaxter,mmehlich}@semanticdesigns.com

Abstract

While a number of research systems have demonstrated the potential value of program transformations, very few of these systems have made it into practice. The core technology for such systems is well understood; what remains is integration and more importantly, the problem of handling the scale of the applications to be processed.

This paper describes DMS, a practical, commercial program analysis and transformation system, and sketches a variety of tasks to which it has been applied, from re-documenting to large-scale system migration. Its success derives partly from a vision of design maintenance and the construction of infrastructure that appears necessary to support that vision. DMS handles program scale by careful space management, computational scale via parallelism and knowledge acquisition scale via domains.

1. Program transformations for software evolution: Transitioning theory into practice

Source-to-source program transformations were originally conceived as a method of program generation in the 1970s [5], and the technology has been developing since [11, 12]. 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 [3, 10]. Theory about how to modify programs transformationally using previously captured design information was suggested in 1990 [6]. 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 is recently popularized [9] as a methodology with suggestions for tool support. Tools for refactoring SmallTalk [14] and Java have started to appear, and some experimental work has been done in refactoring C++ [15]. The better Java tools [16, 17] do some sophisticated refactorings such as ExtractMethod; others in the market require some manual validation of the steps. The most advanced refactoring tool offers a fixed set of 25 transformations selectable from a menu, but these tools are based on conventional, procedural compiler technology solutions, rather than transformations.

Semantic Designs is attempting to make general program transformation systems into a practical tool for software analysis, enhancement, and translation based on a theory of change [7]. The DMS® Software Reengineering Toolkit[1] is a first practical step towards this vision.

Other papers focus on technology and theory of individual mechanisms. We claim the principal issue behind such systems is not technology, but *scale*. We have strived to make DMS scalable as a key to long-term success. (Although [11] is now available commercially, it failed in its original industrial research context partly due to its inability to handle programs of even 10K SLOC). The principal contribution of this paper is showing the set of integrated mechanisms needed to produce a scalable, practical tool.

This paper sketches the DMS vision, describes the scale problems, describes the DMS mechanisms and how they support scale, and examines a number of applications of DMS. This is a lot of material and we are necessarily brief.

2. *Application Scale* as a barrier to tools

Serious progress for practical tool usage requires facing the problem of real systems, *scale*:

- Sheer Size: Real systems of interest are enormous. Many organizations have single applications with 250K SLOC or more, and 40-million line systems are easily found (e.g., Microsoft's Windows OS).
- Multiple Languages: Large software systems typically have multiple languages, including conventional (C, Java, COBOL), scripting (JCL, sh, TCL), domain-specific (SQL, XML, CICS) and customer-defined languages.
- Information spread: Understanding the information flows across the entire system is necessary to make serious changes, whether these flows are interprocedural, interprocess, interlingual, or distributed.
- Design Information Volume: A huge system in essence has a huge specification, and a vast design relationship between the specification and the code. Attempts to change such systems must have access to the specifications, understand why the system is structured the way it is, and have access to a large repository of implementation techniques in the problem domain so that new demands can be met.
- Configurations: a large system may exist at any moment in a large number of variations, due to dif-

[1] DMS is a Registered Trademark of Semantic Designs

Published in *Proceedings, International Conference of Software Engineering*, May 2004.

fering customer needs, execution environments, and decades-long software system lifetimes.

- Teams: Large numbers of engineers are simultaneously engaged in enhancing these systems. Avoiding interactions among activities is a must for progress.

Many tools have been proposed and implemented to take on various aspects of automating program evolution (structure editors, code restructures, source code optimizers, generators). But by and large, these tools at best solve point problems in a single language for at best modest size.

Failure to address scale issues will ultimately doom any automated tool for maintenance, by making it irrelevant to the very systems for which it can provide the most economic value. To build support tools that have a chance of operating on large systems, one must make continual investments in scalable tool infrastructure or risk forever repeating the construction of inadequate infrastructure.

3. The DMS vision: *Design Maintenance*

DMS is a vision of how software evolution can be managed with automated tools, based on these ideas:

1. Software system design can be captured as a formal artifact explaining *what*, *how* and *why*.
2. Desired changes to software in specification, performance, and implementation technology can be explicitly captured as formal *maintenance deltas*.
3. Maintenance deltas can be used to incrementally and mechanically update the design, realizing the desired change while keeping the design current.

First, the fundamental artifact to capture and modify is the *design information* that rationalizes its implementation given its specification [6]. The implementation is an easily extracted part of the design. Such a design must encode:

- *What:* A specification, of both function and performance. Performance addresses all nonfunctional properties of the program. Covering all these issues requires multiple specifications.
- *How 1:* An implementation (source code) in the target languages (typically more than one) that the system has. While this may appear extreme, design information occurs at all levels of abstraction; this is simply the lowest; this is justified in [6].
- *How 2:* A derivation of the implementation from the specification, showing how each functionality specification fragment is correctly realized by code fragments using some implementation technology.
- *Why:* A rationale for each derivation step choice (out of many possible choices) that demonstrates that the actual choice meets the performance requirements.

This design information cannot be practically regenerated on demand, nor can the present chaos of hiding it in large, constantly changing team of software engineers ever be economically practical. Designs are not small; derivations of 10000-line programs can require hundreds of thousands of decisions (number of transformations applied).

Second, the vision suggests encoding of intent to change a system as *maintenance deltas*, covering changes to functionality, performance, and implementation.

Third, the fundamental value of the vision is to use the maintenance deltas to drive incremental changes into the design, achieving both modest change costs and reliable change integration. This can be accomplished by using algebraic distributive properties of the derivation process and design product with respect to the deltas. A complete set of delta propagation algorithms is detailed in [6].

4. The DMS implementation: scalable program transformation infrastructure

DMS is also a running implementation, which is intended to be a scalable stepping-stone to the ultimate vision. The scope of the DMS vision is admittedly quite big, and it is difficult to build this full vision in a short time.

One must start somewhere. A significant portion of the design information is the derivation of the code from the specification. The balance is the rationale as to why a particular derivation was chosen. Thus it is necessary to be able to capture the derivation, and a key step towards doing that is to be able to apply transformations to specifications.

The present version of DMS, the DMS Software Reengineering Toolkit, has been under development for 8 years, and is capable of defining multiple, arbitrary specification and implementation languages (*domains* [12]), applying analyses and both procedurally implemented and source-to-source transformations to source code written in any combination of defined domains (Figure 1). This meets one of the criteria: being able to handle multiple languages in real systems, and applying transformations.

The DMS Software Reengineering Toolkit can be considered as extremely generalized compiler technology, and presently includes the following tightly integrated facilities:

- A hypergraph foundation for capturing program representations (e.g., abstract syntax trees (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 context-free parsers and prettyprinters for arbitrary context free languages in order to convert specification and language instances to and from appropriate internal hypergraph representations.
- Support for defining and updating arbitrary namespaces with arbitrary scoping rules, containing name/type/location information.
- An attribute evaluation system for encoding arbitrary analyses over ASTs.
- An AST-to-AST rewriting engine that understands algebraic properties such as commutativity and associativity. (A hypergraph rewrite engine is planned).

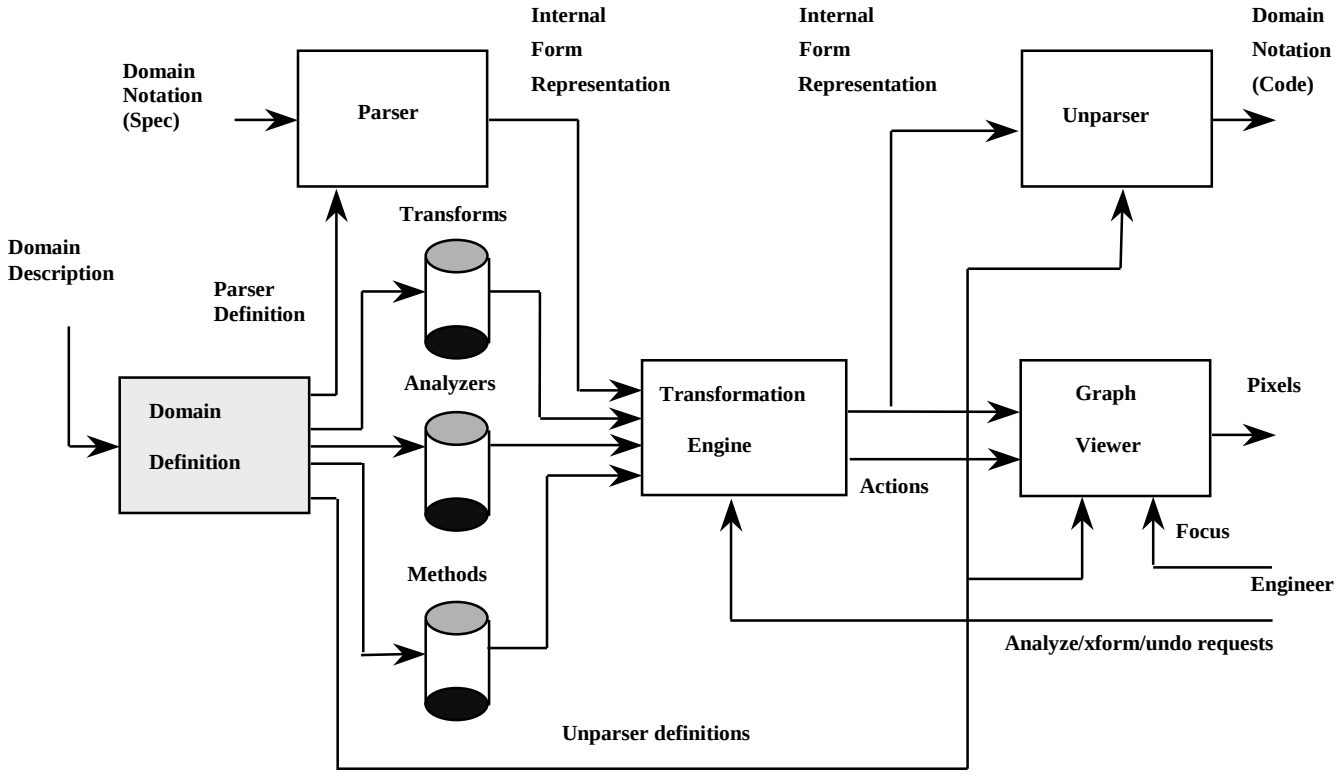

Figure 1: DMS Architecture

- The ability to specify and apply source-to-source program transformations based on language syntax. Such transforms can operate within a language (*optimizations*) or across language boundaries (*refinements*).

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 enabling DMS to run on commodity x86 symmetric-multiprocessing workstations.

DMS as presently constituted has been used for a variety of large-scale commercial activities, including porting, domain-specific code generation and the construction of a variety of "conventional" software engineering tools.

4.1. PARLANSE™ [2]

A fundamental problem is the sheer size of large software systems. DMS is capable of reading literally tens of thousands of source files into a single session to enable analysis across entire systems. Our goal is to handle 10 million lines of code in a typical 4 Gb address space; to date, we have processed 2.5 million Java lines in 2Gb.

Size translates into computational costs: 1 microsecond per source line means 2.5 seconds of CPU to *do anything* to 2.5 million lines. To help alleviate that problem, DMS is implemented in a fine-grain parallel language, PARLANSE [1]. PARLANSE is designed support *irregular parallelism for symbolic manipulation*, induced by large irregular data structures such as million-line ASTs, on shared memory multi-processors.

PARLANSE is the programming language for coding DMS, and for writing DMS application "scripts" which sequence complex series of actions often required to carry out a sophisticated program analysis or transformation. REFINE [22] is similar in intent but not parallel, but the XT composable-tool model [19] is completely different.

PARLANSE is modeled after C, but with LISP-like syntax. It is statically typed, including scalar data types (Booleans, Unicode characters, integers, floats, pointers), arrays, structures, functions, etc. This choice was made on the grounds that when one cannot go parallel, one should be able to execute very efficient serial code, and we wanted to be able to take advantage of the vast experience available in compiling C-like languages efficiently. The present PARLANSE compiler is stand-alone and rather ad hoc due to the need for PARLANSE to exist before any of DMS

[2] PARLANSE is a trademark of Semantic Designs.

could be built. It does not generate particularly good code, but our plan is to replace it with a DMS-based PARLANSE compiler, taking advantage of DMS's growing strength to analyze and optimize large, complex systems, and the ability to specify machine-code generators based on instruction set specifications, an approach which is widely used by the compiler community.

To support software engineering of the big system DMS was expected to be, PARLANSE provides modules, exceptions and downward function closures. Pointer type casts are illegal, which makes PARLANSE strongly typed. Storage management facilities include classic **new** and **free** with non-classic nestable storage pools, which enable the effect of purpose-specific garbage collection. PARLANSE also offers cheap resizable dynamic arrays and (Unicode) strings. A debug-compile assertion checking facility called **trust** allows an engineer to state his expectations of runtime state which are checked during debugging but not for production compiles.

PARLANSE parallelism is based on computational *grain*s, which intend to convey the idea of relatively small chunks of code that can execute in parallel. The fundamental idea is that the PARLANSE programmer specifies explicitly all the parallelism available in his application down to "very fine grain"; the compiler and runtime system choose which parts of the parallelism are realized, sometimes by not generating parallel code, sometimes by throttling parallelism to prevent the runtime system from drowning in lists of available work. This relieves the programmer from the burden of managing the underlying architecture.

Parallelism is delivered in a number of flavors. Dynamic parallelism delivers classic parallelism such as **spawn** (fork with a function applied to a value) and **wait** for grain completion. Classic semaphores and non-classic *future*s (waiting for a cell to acquire an initial value) provide basic synchronization. Because grains may be used to compute speculatively, they may be **abort**ed; grains may catch asynchronous exceptions to clean up before expiring.

Grains may be organized into dynamic self-organizing **team**s. This allows a conceptual computation over a large irregular structure (such as a tree) to be organized as a parallel team that can **recruit** as many grains as needed based on size and structure (often, a recursive call will be implemented by **recruit**). One can **wait** on team completion, and **abort** teams. Exceptions thrown by team members abort other team members; by catching abort exceptions a team can clean up before all members expire. No other parallel programming language known to us offers exception handling cleanly integrated with parallelism.

Static teams declare a fixed number of grains with known styles of interaction. Potential-parallelism (denoted **||**) declares a set of grains, which have no time-ordering relation. Concurrency (**|!|**) declares that the member grains must overlap in time, presumably to communicate with one another. Partial-orders (**;|**) state the partial-order sequencing constraints across a set of grains. Figure 2 describes such a partial order written in PARLANSE consisting of four labeled actions and declarations for execution orderings stating that the first action has to be executed before (<<) the second and fourth ones and the fourth one has to be executed after (>>) the third one. The PARLANSE compiler provides special support and optimizations for these constructs to minimize and inline context-switching overhead.

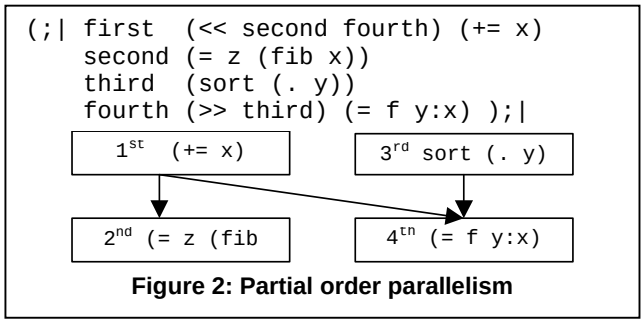

Figure 2: Partial order parallelism

Much of DMS's parallelism can be explicitly specified based on the application of interest. However, a considerable part of DMS's parallelism is automatically extracted from DMS attribute evaluators, which are compiled into several-hundred-thousand-line partial-order parallelism PARLANSE codes. Here, DMS is acting as a code generator to produce significant parts of its own infrastructure. We believe these to be some of the largest parallel programs on the planet, and they are extremely reliable.

DMS presently comprises about 400K SLOC of hand-written PARLANSE code. Several million additional lines of PARLANSE code have been produced by various DMS parts, notably the attribute evaluator generator.

SMP workstations run DMS with significant speedups, but we have not done formal performance studies. There is also still significant room for tuning.

PARLANSE is unique. We know of no other program transformation system built using a parallel language, let alone one that addresses irregular computation. No other parallel programming language integrates exception handling in as robust a way. It has proven its utility in building a system as complex as DMS.

4.2. Hypergraphs and ASTs

The foundations of DMS enables it to represent arbitrary (hyper)graphs, composed of typed nodes carrying literal values (Boolean, character, integer, float or string), and having numbered ports that can represent singleton, sets, or sequences of bi-directional connections to other ports on "neighboring" nodes. Each domain defines its own set of node types, and each node type defines a fixed set of ports having designated connectivity. This representation has been chosen to enable capture of arbitrary graph-like languages, while being implementable in single cache lines, providing for both compactness, to enable large representa-

tions to be captured, and speed of access by avoiding multiple memory fetches once a node becomes of interest, which is important to minimize the memory traffic generated by a parallel graph visit.

At present, DMS provides good support for Abstract Syntax Trees, but it is designed to handle arbitrary graph representations. It is quite easy to encode ASTs on top of hypergraphs: Port 1 is canonically treated as the connection to a (set of) parent node(s) "child" ports. Nodes representing leaf literal values have type specific to the type of the literal, with only a parent port and an associated literal value. For fixed-arity nodes such as "Divide", (child) ports 2 and 3 provide singleton connections to corresponding numerator and denominator subtrees

AST nodes in DMS almost always represent grammar rules and terminal tokens, and so ASTs are typically grammar derivation trees. A number of techniques are used to encode such trees more densely to aid space scaling; because memory accesses are become progressively more costly as processor clock rates rise, so this also aids computational performance. Valueless terminal tokens in the grammar always have node types defined, but may in fact be eliminated from the representations; this often saves a factor of 2 or 3 in space. Long chains of unary productions can be independently eliminated, saving another factor of 2 to 4 due to the deep operator precedence hierarchy in most modern programming and specification languages. Lists are represented by nodes whose port 2 is a sequence of connections to parent ports of subtrees. Ambiguous trees carry a literal designating the token represented, and have a port with a set of children representing alternative trees. In fact, DMS operates on Abstract Syntax DAGs where subtrees can be shared among a forest of trees in a single graph. This often allows ambiguous trees to be stored with maximal sharing to save space. Composing these savings usually results in a significant size reduction of ASTs.

The fact that ASTs are just a special case of a graph makes will make it straightforward to decorate arbitrary graphs with expression trees.

One can manipulate hypergraph nodes procedurally via a PARLANSE Graph module, which provides a parallel-safe way to create and destroy nodes, as well as connect and disconnect their ports, and to read and write arbitrary graphs as text files. A complete AST Interface module for manipulating ASTs is provided, including node creation and destruction, parent/child navigation, connection and disconnection, and operations on complete trees (copy, traverse-with-visitor, destroy, print-as-S-expression).

It is often useful to associate arbitrary structures with graph nodes, including other graph nodes. A standard, parallel-safe high-performance generic hash table provides this facility. Comments and control-flow nodes are associated with AST graph nodes this way.

4.3. Parsing

An extremely practical technology point is DMS's ability to explicitly define languages, parse programs in those languages, and build ASTs for these programs using a methodology applicable to a full range of specification, legacy and modern programming languages, following the Draco domain model [12]. Being able to define sets of such languages quickly and reliably is a necessary scaling property if DMS is to be used to manage large, complex application systems. Given a domain definition, DMS is able to directly read and apply source-to-source transformations in that language.

A DMS parser may be implemented as an arbitrary procedure that produces a hypergraph (this will eventually provide DMS with parsers for graph and visual languages), but all present DMS parsers are implemented using built-in support for integrated lexing, preprocessing, and parsing.

Lexers are defined using a LEX-like scheme for describing sets of regular expressions for whitespace, comments, keywords and value-carrying tokens. Regular expressions include the usual character-sets, Kleene star and plus, and RE union, and the unusual RE intersection and complement. The lexers directly support Unicode but input stream managers can convert from other standard text representations, notably 7 bit ASCII and "European" ASCII ISO-8859-1. Lexical macros allow the definition of named regular expressions, and a large library of Unicode-based character subsets and standard token definitions are supplied as conveniences.

In order to simplify later computations, procedural values attached to each lexeme definition convert values associated with lexemes (integers, floats, characters, strings) to native PARLANSE representations, capturing lexeme shapes ("formats") in the process. Formats describe the syntactic variant of the lexeme, and capture such information as number radix, string-quote-style, keyword case, trailing zero count etc. A powerful conversion library supports conversion to the native representation.

The lexer produces a string of lexemes to be preprocessed and then parsed. Comments are treated specially, and sequences of them are attached as pre-comments or post-comments to the nearest lexeme, depending on domain-definer provided heuristics.

The lexer provides facilities for opening and stacking multiple source streams, providing the basis for preprocessor include files. All lexemes are stamped with source position information including source file, line and column number, enabling accurate reporting of locations for errors or analysis results. Many languages practically require different lexing modes in different parts of the source program, e.g. a main mode for GNU C and another mode for processing embedded assembly language instructions. DMS lexers support the definition of multiple named lexing modes, with procedural actions defining when lexical mode

switches occur. These modes are named so they can be referenced by source-to-source transformation rules. Each mode provides a set of token definitions, which are compiled by DMS into a high-performance finite-state-lexer.

Preprocessing is handled either by procedural attachments to preprocessor token definitions (such as text-string macro definitions as in JOVIAL) or by a separate language-specific preprocessor placed between the lexer and the parser (for C, C++ and COBOL, for which DMS has full preprocessing ability). Generally the strategy is to avoid expanding preprocessor directives, on the grounds that DMS should process "what the programmer sees", not what is seen by the raw compiler. To this end, the preprocessors often carry out partial preprocessing by collecting macro definitions etc, used to drive later preprocessing/parsing, but do not actually expand macros or evaluate conditionals. Instead, these tokens are usually passed on to the parser unchanged. Passing preprocessor directives to the parser require the language grammar be decorated with preprocessor syntax at statistically common places; at points where the grammar will not accept a preprocessor token, the partial preprocessor punts and expands that directive. In practice, this means source programs must be occasionally modified to move a badly placed preprocessor directive to a more convenient location. While this kind of scheme seems unwieldy, a typical DMS user with 1200 C++ files can make the necessary changes to his sources in about a day without breaking them. We are working on a more general scheme that will allow preprocessor directives anywhere.

DMS based parsers take streams of lexemes and parse them according to very simple context-free grammars. All grammar rules are of the form:

LHS = RHS1 RHS2 … RHSn ; *semanticaction*

where *semanticaction* is optional, and is the name of a semantic predicate whose failure disallows a particular LHS expansion. We find the absence of the usual grammar sugar such as Kleene star/plus, alternatives and grouping to be only a small inconvenience; and those constructs would make defining attribute computations more confusing.

We have repeatedly encountered complaints from other researchers about how difficult languages (such as C++) are to parse, and seem to continually find language-processing projects whose goals are lofty but are mired in the mud of achieving a robust parser, thus resulting only in toy tools. Much of this problem comes from choosing parser generators, such as YACC, "found under the lamppost" rather than using very strong technology. DMS uses GLR [21] parsing technology, which generalizes LR parsing by efficiently trying all possible parses in parallel, providing the ability to do full context-free parsing; this also allows the detection and easy capture of ambiguities. In contrast, LALR and other parsing technologies must commit to a particular parse without knowing what is coming next, and often then make the wrong commitment, because extremely few real languages (esp. C++) are in the category of languages parseable by these widely available tools. Consequently all kinds of heuristics and various parser hacks that tangle symbol type collection and lookup into parsing are reinvented to help alleviate these problems. We have found with GLR parsing that we can use a close derivative of the language reference grammar with remarkably modest effort, with the benefit of being able to cleanly separate parsing from non context-free issues such as names and lexical scoping rules. As a consequence, DMS is available with production grammars for an astonishing variety of programming languages, including COBOL, C, C++, Java, SQL, JavaScript, PHP, etc., complete with lexical peccadilloes (weird grammatical syntax, include files, macros, conditionals and dialects), as well as specification languages such as Spectrum, XML, Z, etc.

The DMS parser, which is derived automatically from the grammar, automatically produces a parse tree removing unnecessary tokens and unit productions to produce a compact "abstract" syntax tree. Multiple parses for the same phrase are captured under special *ambiguity* nodes, which share subtrees to save space. Such ambiguities are usually removed by a symbol-table construction step that follows parsing. However, the parsing process can use domain-engineer-provided semantic constraints on reductions, and these are occasionally used to eliminate ambiguities while parsing. This works extremely well for FORTRAN, in which line numbers on statements can disambiguate loop nesting/overlap, thus allowing the parser to produce ASTs with proper loop nesting, as it parses. The parser also captures lexeme values and comments and attaches them to the appropriate terminals so that a resulting parse tree is a complete model of the source text. This enables later regeneration of full text after transformation is complete.

The XT project [19] also seems to have discovered the utility of GLR parsing, and appears to have similar success in parsing a variety of languages, although we have not seem claims of full parsers for production languages. However, XT goes to extremes by parsing the source text at the character level; they do not use a lexer. This is conceptually cleaner than DMS's traditional lexer/parser approach, but we do not believe it is as efficient in practice. It also leaves the difficult problem of handling comments; they must be produced as part of the derivation tree in a character-level parsing regime, yet they can occur anywhere without providing any semantics. Consequently manipulating such a character-oriented tree is more difficult. In practice, what XT tools appear to do is to process the derivation parse tree through an abstract syntax tree builder, which provides the opportunity to move the comments to somewhere more convenient. But this simply brings the comment problem back; going to character-level parsing doesn't fix it.

4.4. Attribute evaluation

Change requires knowing where to change, which in turn requires analyses to be performed. DMS provides analysis support in the form of attribute grammars defined over the domain syntax, for arbitrary subtrees. This allows encoding of analyses, which can be relatively easily expressed in terms of the language structure using inherited and synthesized attributes and value-combining operations. Examples of information computed easily this way include metrics, set-of-operands, control flow, symbol tables, etc. Analysis computations not fitting this model can be implemented as arbitrary PARLANSE code.

Traditional attribute evaluators are purely functional; DMS attribute evaluators also allow side effects (procedurally specified in PARLANSE), which are extremely convenient for updating large analysis results rather than passing them around. Two common "large" values are sets (implemented as PARLANSE dynamic arrays) and symbol tables for complete software systems.

Attribute computations are compiled into PARLANSE code, one procedure per grammar rule. Because the attribute evaluation information flow within a rule is essentially functional (and side effects are explicitly stated), it is relatively easy to compute partial orders over the computations defined in each attribute rule. Consequently each rules' attribute computation is mapped straightforwardly onto a PARLANSE partial order. Thus attribute evaluation in DMS is irregularly parallel, driven exactly by the shape of the desired computation. We believe that the attribute evaluators we produce are likely the largest parallel programs in existence, and they are reliable because they are synthesized. While there have been a number of parallel attribute evaluators implemented experimentally, we think DMS's is one of the first to be practical on a large scale.

Each attribute evaluator is made available as a PARLANSE function so it can be called by other arbitrary procedures, including other attribute evaluators.

We are working on generic infrastructure for information flow analyses based on control flow analysis. We presently extract control flow graphs by attribute evaluation.

4.5. Symbol table support

Only very simple notational systems are context free. Systems used for practical purposes (C++, Verilog, XML…) all have rules for naming entities and complex scoping systems to manage the huge namespaces that occur in practice. One cannot realistically implement transforms on such notational systems without providing a means for managing the discovery of name definitions and name lookups: symbol tables.

DMS provides a general symbol table management subsystem with facilities for defining and recording name/type information associations in symbol spaces, where names are arbitrary Unicode strings and types are arbitrary PARLANSE structures (including possibly references to other symbol spaces) defined by hand per domain to represent the types in that domain. Symbol spaces are implemented as a scalable, parallel-safe-access hash table to enable parallel PARLANSE computations to access and update them reliably.

Symbol spaces have parent-links with associated integers (Figure 3). A symbol table is a set of symbol spaces with established parent arcs. A standard lexical lookup on a symbol searches from a designated starting space, through parent spaces in integral order until the symbol is found in some space or all paths to parents are exhausted. Multiple parents with integer priorities make it easy to implement features like multiple-inheritance. The notion of "match" is defined by per-domain PARLANSE Boolean functions provided as parameters to the search, enabling straightforward lookups, even with "overloading" scoping rules.

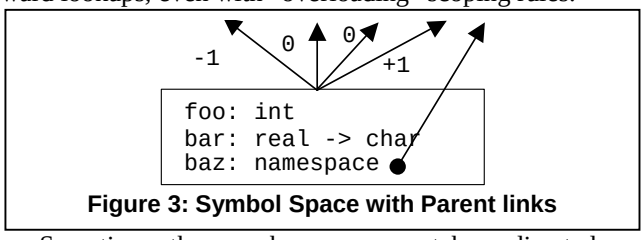

Figure 3: Symbol Space with Parent links

Sometimes the search process must be redirected according to the symbol space from which it emanates. Different types of symbol spaces accomplish this, including one, which allows an arbitrary action in the middle of a search. This enables the implementation of Java rules that search the file system if a name cannot be found in a particular scope.

A symbol table instance is commonly built by an attribute evaluator, with symbol space references flowing around the tree, and side effects inserting new entries in the symbol spaces. Because of the partial order evaluation, symbol table construction for a set of files automatically occurs in parallel, which helps when processing a large system. We have done this for C, C++, Java, JOVIAL, and COBOL. The Java version runs parsing and symbol table construction in parallel as it processes source files.

Typically a domain-specific API is constructed on top of the symbol table machinery with facilities for looking up an identifier in a context (e.g., lookup the identifier in this tree node) and for updating the symbol table. This supports carrying out program transformations.

Remarkably, pure rewrite transformation systems like TAMPR, TXL and XT try to avoid providing symbol table support. This forces their users into coding transforms to implement lookups, which is extremely inefficient at best, and clumsy to code and maintain at worst. They succeed in theory because rewrites are Post systems and therefore Turing capable; but in practice, nobody wants to program a Turing machine. Pure strategies aren't necessarily good engineering strategies.

4.6. Transforms and source rewrite rules

Program transforms are generally described as source-to-source rewrite rules. In theory, however, transforms are simply functions from program representations to program representations.

DMS offers both views in practical forms and the tool engineer mixes and matches as convenient. One often finds procedural transforms using source transforms as support, and source transforms using procedures (often attribute evaluators) to check complex applicability conditions and generate right-hand-sides.

Extensive but straightforward APIs are provided to allow direct procedural manipulation directly of the hypergraph, or more conveniently, of the abstract syntax trees using PARLANSE code. Most compiler infrastructures stop at this point. So do many program manipulation tools such as OpenC++ [20]. However, this forces the tool engineer to become intimately familiar with the microscopic details of the tree representation, which is a burden for languages the size of C++ and for new languages whose structure is still evolving. It also means the transform rules cannot be inspected by anybody but the authors.

DMS offers source-to-source rewrite rules stated in the domain notation of interest. A typical rewrite rule abstractly has the following form:

LHS → *RHS* **if** *condition*

where both LHS ("left hand side") and RHS ("right hand side") represent source language patterns with variables to represent arbitrarily long well formed language substrings. The **if** condition is an optional phrase referring to the variables in the LHS pattern. These rules are interpreted as, "when a program part matches the LHS, replace it by the RHS, if condition is true". The condition may be implemented as some additional matching constraints, or a call on some decision procedure coded in PARLANSE.

```
default domain C.
rule auto_inc(v:lvalue):
  statement->statement =
  "\v = \v+1;" rewrites to "\v++;"
  if no_side_effects(v).
```

Figure 4: A DMS rewrite rule

Real transformation systems add more syntax to this simple scheme to allow specification of more details about the patterns. For DMS, an example C source code rewrite to convert an assignment into an auto-increment is shown in Figure 4; its effect is demonstrated in Figure 5. This rule is written in DMS's Rule Specification Language. The domain phrase tells the transform tool to interpret following rule text inside quotes as being C syntax with escapes for meta-variables such as **v**. (We have taken slight liberties with the transforms to simplify their presentation). The DMS parsing machinery provides pattern-parsing capabilities derived automatically from the parser-description for the domain, and converts such phrases into pattern trees.

```
before: (*Z)[a>>2]=(*Z)[a>>2]+1;
after:  (*Z)[a>>2]++;
```

Figure 5: Result of Applying Transform

Finally, reasoning is based on mathematical systems involving algebraic formula rewriting. DMS provides a tree-rewriting system capable of handling associative and commutative laws, thus providing a basis for rationale generation and capture. The rewriting engine does double duty as the basis for the tree-to-tree rewriting implementation of the program transforms.

Typically a transformation system will have a large number of rules, and a large number of possible places in a program to apply them. It is beyond the scope of this paper to describe how the transformation system chooses which rules and where to apply them.

4.7. Prettyprinting

Having transformed a parse tree to obtain an improved tree, or having synthesized a new tree, it is eventually necessary to convert it back to a text representation.

DMS provides a pretty-printing facility, based on the notion of constructing and composing text boxes. Associated with each grammar rule is a specification of how to regenerate text for a tree node representing the rule nonterminal in terms of box operations applied to text boxes produced for children's tree nodes. Box operations include:

- *Primitive Boxes* are produced by procedures defined for terminals, with many defaulting to built-in DMS primitives. These typically handle reconstruction of keywords, text for numbers shaped by the format information stored with the AST node, etc. Custom procedures can produce interesting results, such as substituting nonsense identifiers for actuals (giving code obfuscators) or HTML hyperlinks to source positions of identifier declarations (producing hyperlinked cross reference documents).
- **`H(…)`** Horizontal Concatenation. Assembles two text rectangles into a larger rectangle by left-right juxtaposition.
- **`V(…)`** Vertical Concatenation: Top/bottom box juxtaposition.
- **`I(…)`** Indent. Adds whitespace on the left of a box.
- **`if then else endif`** conditionals, to allow multiple dynamically-selectable prettyprinting styles to coexist.

Prettyprinting rules are written using grammar terms. The grammar rule and corresponding prettyprinter rule:

```
if = 'if' expression 'then' stmts 'end';
  <<PrettyPrinter>>:  {
    V(H('if',expression,'then'),
      I(stmts),'end'); }
```

are sufficient for DMS to both parse and regenerate a nicely indented if-then-endif block. Comments attached to AST nodes are automatically reinserted at the appropriate place, with indentation matching the current indent-distance.

The DMS prettyprinter can operate in "prettyprinting" mode or "fidelity" mode. In prettyprinting mode, it honors the supplied prettyprinting rules. In fidelity mode, it reproduces the spacing implied by the source position found attached to each tree node; if the source position is null, it falls back on prettyprinting rules. This allows the prettyprinter to reproduce familiar program text where transforms have not been used, and to produce readable text where transforms have introduced new code.

Prettyprinters are implemented as a special case of attribute evaluation in which the internal state of the pretty printer is passed around the tree.

5. Applications of DMS to software evolution

The present DMS has been successfully used for a number of commercial tasks:

- Generation of domain tools. A good test for the DMS infrastructure has been using DMS to implement much of DMS. Domain-specific languages for lexing, grammars, prettyprinting, attribute evaluation, and program transformations are all implemented using DMS. Further DMS subsystems are expected to use DMS aggressively for the conceptual clarity and performance one can get from specifying and applying complex implementation transforms, thus increasing the scale on which DMS can operate.
- Automated detection of code clones, in Java, C, and COBOL, on systems of 500K SLOC to 2.5M SLOC [8]. For C and COBOL, transformations have been applied to remove clones. We are presently working with a commercial customer to evaluate the use of clone detection as a source of domain concepts in large scale Java applications.
- Simplification and removal of C/C++ preprocessor directives based on partial evaluation of preprocessor conditionals [18].
- Code generation of factory controller programs from factory process specifications. These programs are in experimental use in US automobile factories. DMS's algebraic rewriting facility enabled high degrees of optimization of the thousands of large Boolean equations typically generated.
- Implementation of production test coverage and profiling tools for several dialects of C, C++, Java and COBOL [2]. These tools have been used on software systems with 4000 files.
- Synthesis of compact, extremely fast Java parsers for specific XML DTDs.
- Translation of large-scale real-time JOVIAL source code programs for 16 bit minicomputers to 32 bit C-based programs. An existing flight application of 370K SLOC has been automatically translated, preserving code, comments, and macros, and is presently undergoing ground-system tests.

Ongoing research work using DMS includes:

- Restructuring of Web sites
- Restructuring a custom distributed C++ system with 6000 components to change the component architecture to be compatible with CORBA.
- Pushing model-driven aspects into the corresponding modeled C++ code.

Scalability as a driving concern during DMS design has been a principal contributor to DMS's ability to carry out most of these tasks. This scale capability has in turn made DMS commercially effective, providing fuel for further development.

6. Where to next?

We are still a long way from being able to carry out complex analyses on large systems. Most of what is needed here is the basic flow analysis infrastructure long used by optimizing compilers. There is active work at SD to implement such generic infrastructure.

Working with real systems requires a number of domain languages be pre-defined for use. Among others we presently have Java, C, C++ and COBOL85 well in hand. Other real languages require someone to provide their definition; as a practical matter, we do these as commercial opportunities arise. One dismaying aspect is the number of dialects of such languages. While there are standards for the mainstream languages, no vendor actually implements that standard exactly, and the syntactic and semantic differences require additional attention (DMS provides a "dialect" configuration management scheme to help cope with this). Non-mainstream languages are much worse in this regard; often there are at best untrustworthy language definitions and worse still there tend to be more dialects because of the absence of standards.

Having succeeded with building basic infrastructure, the major tasks ahead to enable Design Maintenance are:

- Defining a number of interesting performance specification languages as domains. We have done promising internal experimentation using algebras. But just like the number of "functionality"-specifying languages, the number of performance specification languages is expected to be large, and there is not a lot of experience in writing these down. Quality Of Service (QoS) is just one example class.
- Defining large libraries of optimization and implementation knowledge as source-to-source transforms where practical, and as procedurally implemented transforms otherwise. These libraries provide potential transformational capability.

- Implementing a transformational "strategy" subsystem, to control transformation rewriting according to performance specifications
- Capturing the transformation steps and their dependences. This is relatively straightforward for source-to-source rewrites because they can be directly inspected. This is harder for procedural transformations, because their effect is opaque; we expect to annotate these to alleviate this problem.
- Providing facilities for displaying and modifying the transformational strategy and derivation steps.

A major issue which we have not explored is how to manage teams of engineers, all trying to inspect/modify such designs, for multiple configurations. This will require long-term transactions applied to designs, in which "atomic" updates to significant parts of a design may take long periods of analysis and operator interaction time.

7. Conclusion

DMS is both a grand vision and attempt to validate that vision, and a practical tool for carrying out large-scale software analysis, modification, and enhancement. The DMS vision dictates how the implementation must evolve, and Semantic Designs is pursuing the implementation of the long-term vision.

The present DMS has taken some 50 person-years of effort over 8 elapsed years to build. We expect that implementing the remainder of the vision will take an equal amount of effort. This is not a small-scale research project! However, we strongly believe that the only way to achieve large-scale system evolution capabilities is to build tools something like DMS, and there is no avoiding the vast effort it takes to put the necessary infrastructure in place.

We remark that this kind of infrastructure is necessary if automated software engineering research is to spend its energy on research rather than reinventing poor versions of pieces of such a tool. Big Software Engineering needs to act like big Physics in terms of infrastructure.

8. Acknowledgements

We dedicate this paper to the late Chris Pidgeon. He shares much of the credit for making DMS possible. Michael Mehlich has been personally responsible for significant parts of the implementation, its C++ domain, and, most importantly, keeping our theory vision high and clean. Sanjay Bhansali, Sheila Cheng, Warren Li, Andrew Yahin and Aaron Quigley all contributed to early versions of PARLANSE and DMS. Leo Moura, Marcelo Sant'Anna, and Srinivas Nedunuri worked on the CloneDR. Hongjun Zheng has implemented many standard SE tools using DMS and its Java domain. Many thanks to Jim Neighbors for opening our eyes to the ideas of domains and program transformations. Lastly, thanks to all the researchers whose various systems provided us the foundation ideas to integrate.

References

[1] *PARLANSE Reference Manual*, Semantic Designs, 1998.
[2] www.semdesigns.com/Products/TestCoverage/index.html
[3] G. Arango, I. Baxter, C. Pidgeon, P. Freeman, "TMM: Software Maintenance by Transformation", *IEEE Software* 3(3), May 1986, pp. 27-39.
[4] R. Balzer, "A 15 Year Perspective on Automatic Programming", *IEEE Trans. Software Engineering* 11, Nov. 1985, pp. 1257-1268.
[5] R. M. Balzer, N. M. Goldman, and D. S. Wile, "On the Transformational Implementation Approach to Programming", in: *Proceedings of the 2nd International Conference on Software Engineering*, Oct. 1976, pp. 337-344.
[6] 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.
[7] I. Baxter, "Design Maintenance Systems", *Communications of the ACM* 35(4), 1992, ACM.
[8] I. Baxter, et. al,. "Clone Detection Using Abstract Syntax Trees", in: *Proc. International Conference on Software Maintenance*, IEEE, 1998.
[9] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison Wesley 1999.
[10] W. L. Johnson and M. S. Feather, "Using Evolution Transforms to Construct Specifications", in: M. Lowry and R. McCartney (eds.), *Automating Software Design*, AAAI Press, 1991.
[11] 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.
[12] J. Neighbors, "Draco: A Method for Engineering Reusable Software Systems", in: T. Biggerstaff and A. Perlis (eds.), *Software Reusability*, ACM Press 1989.
[13] 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.
[14] D. Roberts, J. Brant, R. Johnson and W. Opdyke, "An Automated Refactoring Tool", in: *Proceedings of ICAST '96: 12th International Conference on Advanced Science and Technology*, Chicago, Illinois. April, 1996.
[15] L. Tokuda and D. Batory, "Evolving Object Oriented Designs with Refactoring", in: *Proceeding, Conference on Automated Software Engineering*, IEEE, 1999.
[16] www.intellij.com. IDEA refactoring tool for Java.
[17] www.instantiations.com. Jfactor refactoring tool for Java.
[18] I.D. Baxter and M. Mehlich, "Preprocessor Conditional Removal by Simple Partial Evaluation", in: Wor*king Conference on Reverse Engineering*, pp. 281-290, 2001.
[19] E. Visser, "Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT 0.9".
[20] S. Chiba, "A Metaobject Protocol for C++", in: *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1995. pp. 285-299.
[21] M. Tomita, *Efficient Parsing for Natural Languages – A fast Algorithm for Practical Systems*, Kluwer Academic Publishers, 1986.
[22] Reasoning Systems, Palo Alto, CA, "Refine Language Tools", 1993.