keynote: Software Language Engineering 2013

Dark Knowledge and Graph Grammars in Automated Software Design

Don Batory[1], Rui Gonçalves[2], Bryan Marker[1], Janet Siegmund[3]

[1] University of Texas at Austin, Austin, TX 78712 USA
batory@cs.utexas.edu, marker@cs.utexas.edu
[2] Universidade do Minho, Braga, Portugal
rgoncalves@di.uminho.pt
[3] University of Passau, Germany
feigensp@ovgu.de

Abstract. Mechanizing the development of hard-to-write and costly-to-maintain software is the core problem of automated software design. Encoding expert knowledge (a.k.a. *dark knowledge*) about a software domain is central to its solution. We assert that a solution can be cast in terms of the ideas of language design and engineering. Graph grammars can be a foundation for modern automated software development. The sentences of a grammar are designs of complex dataflow systems. We explain how graph grammars provide a framework to encode expert knowledge, produce correct-by-construction derivations of dataflow applications, enable the generation of high-performance code, and improve how software design of dataflow applications can be taught to undergraduates.

1 Introduction[4]

Like many of you, I read popular science articles. I especially enjoy discussions on current problems in theoretical physics. My favorite problem is that roughly 80% of the mass of our universe is made of material that scientists cannot directly observe. It is called *dark matter*. Dark matter emits no light or energy, but is not entirely invisible. Scientists know it exists because with it they can explain the otherwise unusual rotations of galaxies, the unexpected bending of light in empty space, and the surprising fact that the expansion of our universe is accelerating. The issue of dark matter has been known for at least 25 years [2], yet today it remains poorly understood.

Dark matter reminds me of a corresponding problem in software design. Software design is a series of decisions whose effects are seen in programs, but are not directly observable. In analogy to dark matter, I call it *dark knowledge*. Dark knowledge is fleeting. Programmers may know it one day and forget it the next. It is not present in source code. Yet we know dark knowledge exists, because with it we can explain program designs. If an engineer makes a certain decision, (s)he would expect to see algorithm α in a program; with an alternative choice, (s)he would see β. The presence of dark knowledge in programs has been known for at least 30 years [6,9,22], and today it too remains poorly understood.

[4] As this paper transcribes a keynote presentation, "I" refers to Batory's personal experience and "We" refers to the experience of all authors.

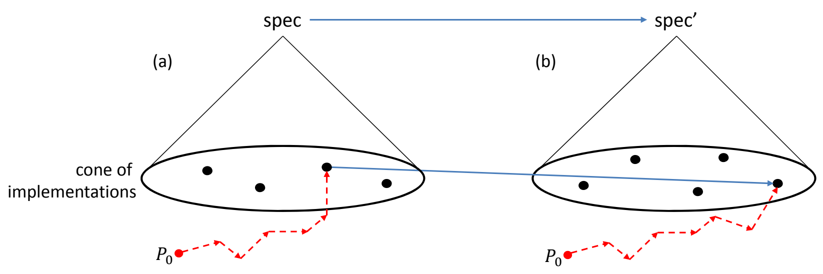


Fig. 1. Cone of Implementations for a Specification.

Dark knowledge is important. Software design starts with a formal or informal specification. We know that there are huge numbers of possible programs that could implement a spec (indicated by the "cone of implementations" in Fig. 1a, [3]). With domain and software-engineering knowledge, an engineer makes a series of decisions to create a program to implement the spec. The dashed lines in Fig. 1a indicate "dark knowledge": the engineer starts with an existing program (possibly the empty program) P_0 that typically does not satisfy the spec, makes a series of modifications (one per decision) to ultimately arrive at a program that does satisfy the spec. This chain of decisions is fleeting—over time it is forgotten, thereby losing vital knowledge about the program's design. When the spec is updated (Fig. 1b), engineers who maintain the program effectively have to recreate the series of decisions that lead to the original program, erase a previous decision, and replace it with those that are consistent with the new spec. In time, these decisions are forgotten as before, bringing us back to square one where design knowledge is dark.[5]

The connection to language design is immediate in a shallow way: language design and language implementation are instances of these ideas. They too involve a series of decisions whose effects can be seen, but not explicitly encoded. Consequently, the same problems arise: vital design knowledge for maintenance and evolution is lost.

The importance of dark knowledge is well-known. Making dark knowledge white (explicit) was expressed in 1992 by Baxter in his paper on "Design Maintenance Systems" [6]. More recently, another version of this idea arises in self-adaptive and self-managing software [48] . Here, the goal is to encode design decisions explicitly in software to make dark knowledge white, so these decisions can be revisited and redecided automatically (possibly with a human in the loop). Illuminating dark knowledge embodies a new approach to software development [40].

The approach presented in this paper to make dark knowledge white is called *Design by Transformation (DxT)* [17,21,37,38,46,50].

2 How Dark Knowledge Can Be Encoded

The challenge is how to encode dark knowledge thereby making it white. Transformations can do this. Fig. 2 shows the basic idea. Starting with program P_0, a series of transformations $\tau_4 \cdot \tau_3 \cdot \tau_2 \cdot \tau_1$ is applied to produce the desired program P_4. Of course, today each of these transformations is accomplished manually: P_0 is hacked into P_1, P_1

[5] Dark knowledge can be encoded in code comments, but this is woefully inadequate.

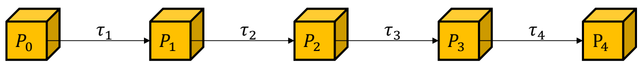


Fig. 2. Program Derivation.

is hacked into $\mathtt{P}_2$, $\mathtt{P}_2$ into $\mathtt{P}_3$, and $\mathtt{P}_3$ into $\mathtt{P}_4$. With enough experience, an engineer can skip intermediate steps. But if each transformation were programmed so that it could be applied by a tool, $\mathtt{P}_4$ would be automatically derived from $\mathtt{P}_0$. It is this automation mindset that drives DxT.

Program derivation in DxT is related to grammars and parse trees. A grammar $\mathcal{G}$ for a language is a set of rules called *productions* that describe how to form sentences (in the language's alphabet) that are valid according to the language's syntax. The set of derivable sentences is the language $\mathcal{L}(\mathcal{G})$. Fig. 3a shows grammar $\mathcal{G}$ and its cone of sentences $\mathcal{L}(\mathcal{G})$. What I have called dark knowledge is a sequence of decisions that derives a particular sentence $\mathtt{S}$ (the dashed arrows in Fig. 3a). Starting from a representation $\mathtt{S}_0$ that likely does not belong to $\mathcal{L}(\mathcal{G})$, a series of decisions (production invocations) derives $\mathtt{S}$. This derivation, of course, is the parse tree of $\mathtt{S}$: it is a proof that $\mathtt{S} \in \mathcal{L}(\mathcal{G})$ (Fig. 3b). It also represents the not-so-dark knowledge of $\mathtt{S}$. Characteristic of dark knowledge is that there is no direct evidence of these productions in $\mathtt{S}$ itself; all that appears are their after-effects. Such knowledge is important; given the *abstract syntax tree (AST)* of a program, one can automate program manipulations, such as refactorings. Without such knowledge, refactorings would be difficult, if not impossible, to perform correctly.

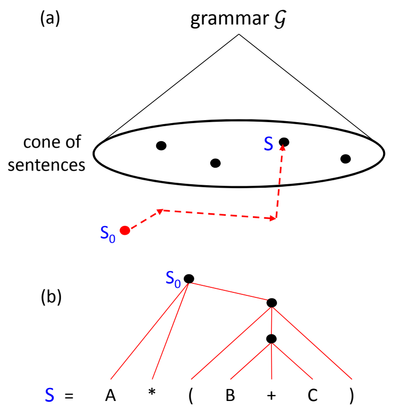


Fig. 3. The Language of a Grammar and a Parse Tree of Sentence $\mathtt{S}$

In over 25 years of studying program design, I have come to see typical programming languages and their grammars as one-dimensional; their sentences (*eg* Java programs) are simply 1D strings. This is not enough: to see the possibilities that arise in program development, one has to think in terms of $n \geq 2$ dimensional graphs, not 1D lines.

I focus on dataflow programs in this paper. They are not representative of all programs, but they do occupy a significant group of programs that are developed today. A dataflow program can be visualized as a graph of computations, where nodes are primitive computations and edges indicate the flow of data. Fig. 4a is an example: α, β, γ are computations; data enters on the left and exits on the right. Although it is always possible to map dataflow graphs to 1D programs, there is an important distinction between 1D and nD grammars, which I'll discuss later in Section 5.1.

Informally, *graph grammars* are generalizations of Chomsky string grammars. They extend the concatenation of strings to a gluing of graphs [10,15]. Productions are of the form $\mathtt{Graph}_{\mathtt{left}} \rightarrow \mathtt{Graph}_{\mathtt{right}}$; *ie* replace $\mathtt{Graph}_{\mathtt{left}}$ with $\mathtt{Graph}_{\mathtt{right}}$.[6]

[6] There are different formalisms for graph grammars [47]. DxT grammars follow the algebraic (double-pushout) approach to (hyper-)graph grammars.

Derivations are of the form $\texttt{Graph}_{\texttt{initial}} \Rightarrow^* \texttt{Graph}_{\texttt{final}}$; *ie* apply a sequence of rewrites to $\texttt{Graph}_{\texttt{initial}}$ to produce $\texttt{Graph}_{\texttt{final}}$. Graph grammars have been used for many purposes, such as design of visual languages [19,44,59], model synchronization [20,28], model validation [25,52], program compilation [49], and dynamic adaptation/evolution of architectures [11,14,33,55,56].

Fig. 4 shows three rewrites. Fig. 4b replaces a β computation with a graph of computations (*eg* a map-reduce of β). Fig. 4c shows the same for γ. Fig. 4d shows that of α followed by α^{-1} cancels each other, yielding an identity map. Fig. 5 is a derivation that starts at an initial graph where computation β precedes γ to the final graph of Fig. 4a using the rewrites of Fig. 4b-d.

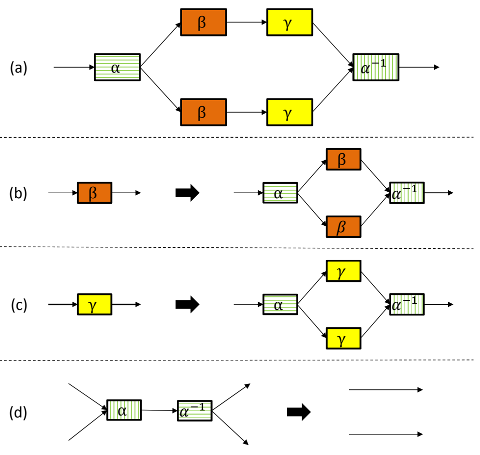


Fig. 4. A Dataflow Graph and 3 Rewrites

A direct analogy of 1D and *n*D grammars would have both defining the syntax of a language. For example, it is easy to imagine a language of cyclic graphs, where each node is connected to exactly two other nodes. DxT goes further in that each production defines a semantic equivalence between its LHS and RHS graphs.

There is a subtle distinction between a graph grammar and a graph-rewriting system: the former enumerates all graphs from some starting graph and the latter transforms a given state (host graph) into a new state [57]. In this sense, DxT is closer to a graph grammar.

All of this is rather abstract, so let's pause here to see a concrete example.

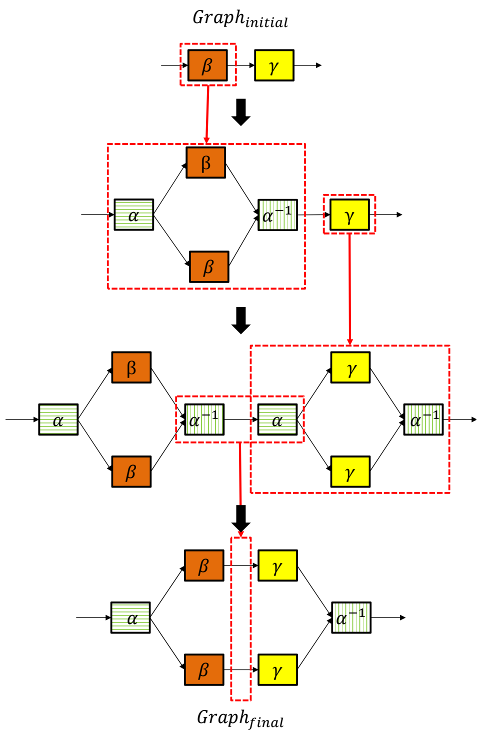


Fig. 5. $\texttt{Graph}_{\texttt{initial}} \Rightarrow^* \texttt{Graph}_{\texttt{final}}$

3 Upright: A Synchronous Crash Fault Tolerant Server

Upright was the state-of-the-art Byzantine crash fault tolerant server in 2009 [8]. We were interested in its dataflow design. Talking to the Upright authors, we soon discovered that ∼15 people on earth really understood it (and we certainly were not among them). It posed a challenging reverse engineering task [46]. In this section, we review Upright's *Synchronous Crash Fault Tolerant (SCFT)* design in terms of DxT. Doing so turns its dark knowledge white.

Upright's starting dataflow graph is Fig. 7a. (My apology for the size of this figure; it can be digitally enlarged). Such a graph in *Model Driven Engineering (MDE)* is called a *Platform Independent Model (PIM)*. Clients (the `C` boxes) asynchronously send requests to a stateful server (box `VS`); the network effectively serializes these requests (box `Serialize`). The server reads each request, updates its state, and then sends out a response. The network routes the response back to the originating client (box `Demultiplex`). In effect, messages exiting on the right of Fig. 7a re-enter the figure on the left, as if the graph were embedded on the surface of a cylinder (Fig. 6).

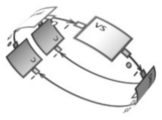


Fig. 6. Cylinder

The derivation of Upright's implemented dataflow graph, called a *Platform Specific Model (PSM)*, begins with the transition from Fig. 7a to Fig. 7b that exposes a network queue (`L`) in front of the server (`S`). Next, the transition from Fig. 7b to Fig. 7c effectively performs a map-reduce of both `L` and `S` [29]. Fig. 7d is a copy of Fig. 7c that shows the subgraphs to be replaced (to eliminate single points of failure). The SCFT dataflow design of Fig. 7e is a PSM for Fig. 7a [46]. We used this derivation to reimplement Upright's SCFT design [46].

The semantics of these rewrites are well-understood by experts of SCFT design; for this presentation, we view them as sterile graph transformations.

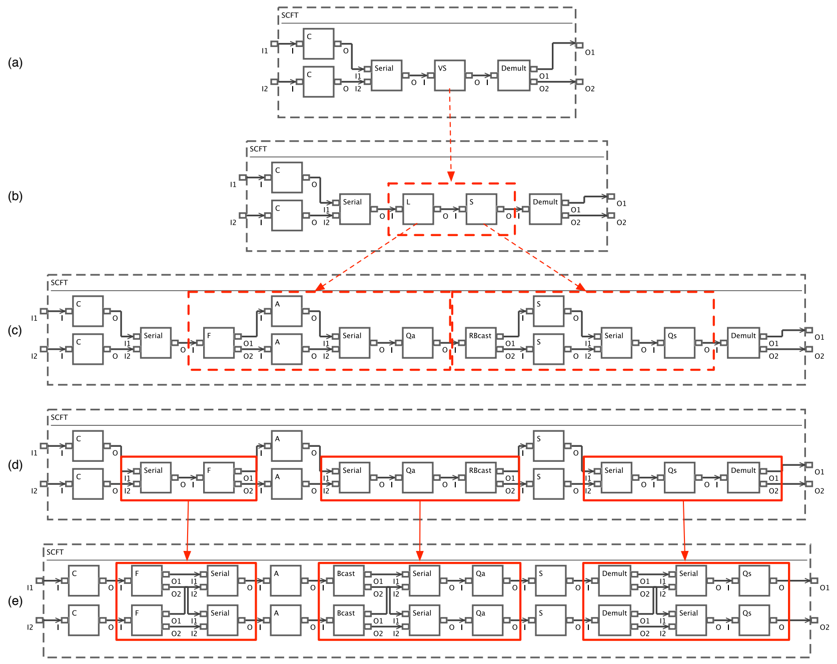


Fig. 7. Upright's SCFT PIM $\Rightarrow^*$ PSM Mapping.

3.1 DxT and the Essence of Graph Grammars

A graph grammar $\mathcal{GG}$ is an ordered pair $(g, \mathcal{P})$; g is the starting graph and $\mathcal{P}$ is a set of graph productions. The language of $\mathcal{GG}$, $\mathcal{L}(\mathcal{GG})$, is the set of graphs that can be derived by applying the rules in $\mathcal{P}$ to g [10,15,47].

DxT builds on this foundation: (1) the primitive computations (operations) of a domain are the alphabet of $\mathcal{GG}$, (2) the fundamental computational equivalences of the domain are its graph transformations (which encode the fundamental patterns of computation that were previously dark knowledge), and (3) the initial graph g is the PIM of an application and $\mathcal{L}(\mathcal{GG})$ is the set of its PSMs—the cone of implementations for g. DxT goes further, in that the initial graph can be a member of a domain of PIMs, $g \in \mathcal{G}_{\texttt{pim}}$.

There are indeed distinctions between 1D and nD grammars. Here are a few:

- In general, the parse of a sentence in a 1D grammar should be unique; the grammar is either unambiguous or it is ambiguous with context making a parse unambiguous. Not so for nD grammars: multiple parses of a dataflow program simply means there are multiple equivalent ways of deriving that program—a perfectly acceptable situation.
- 1D productions do not need proofs of correctness—they simply define a textual pattern where there is nothing to prove. In contrast, each DxT rewrite defines a fundamental computational equivalence in a domain; there should be some evidence (ideally a proof) that each rewrite is correct.
- A parse tree for sentence $\mathcal{S}$ in a 1D grammar $\mathcal{G}$ is proof that $\mathcal{S}$ is a sentence of $\mathcal{L}(\mathcal{G})$. A derivation tree for dataflow application $\mathcal{S}$ in an nD grammar $\mathcal{GG}$ is a proof that $\mathcal{S} \in \mathcal{L}(\mathcal{GG})$, *ie* $\mathcal{S}$ is a correct-by-construction implementation of g.
- 1D technology aims at parsing sentences. Although DxT can also be used for reverse engineering (parse the design of a legacy application), here we use it to derive programs (and explore the space of implementations of a spec).

It is not difficult to imagine the utility of Upright's DxT explanation. I could go into more technical details about DxT now, but that would be overkill. Instead, a big picture is needed to motivate this general field of research, which I consider next.

4 Who Cares? Motivations from Practice

Software Engineering (SE) largely aims at techniques and tools to aid masses of programmers whose code is used by hoards—these programmers need all the help they can get. At the same time, there are many domains where programming tasks are so demanding that there are only a few programmers that can perform them—these experts need all the help that they can get, too.

As said earlier, the focus of my research group is on dataflow domains which represent an important class of today's applications (*eg* virtual instrumentation [53] and applications of streaming languages [54]). The specific domains of our interest include parallel relational join algorithms [12], crash fault tolerant file servers [8], and distributed-memory, sequential, and shared-memory *Dense Linear Algebra (DLA)* kernels [37,38].

In practice, domain experts magically produce a *big bang* design: the dataflow graph of the complete application. Typically, it is a spaghetti diagram. *How it was created* and *why it works* are mysteries to all but its authors. For academic and practical reasons, it seems intuitively better to derive the graph from domain knowledge; doing so would answer both questions.[7] A digitally enlargeable Fig. 8 shows a DxT derivation of the parallelization of hash joins in the Gamma database machine [12]. Ask yourself: would you want only $\texttt{Gamma}_{\texttt{final}}$ or its derivation $\texttt{Gamma}_{\texttt{initial}} \Rightarrow^* \texttt{Gamma}_{\texttt{final}}$? I return to this point in Section 6.

Our current project focuses on the generation of DLA kernels/libraries. Kernel portability is a serious problem. First, porting may fail: kernels for distributed memory (where communication between cores is explicitly handled via a high-speed network [38]) may not work on sequential machines and vice versa. Second, if it does work, it may not perform well. The choice of algorithms to use on one hardware architecture may be different from those to use on another. One cannot simply "undo" optimizations and apply others—hopefully the reason for this is clear: such changes require dark knowledge. Third, in the worst case (which does frequently happen), kernels are coded from scratch.

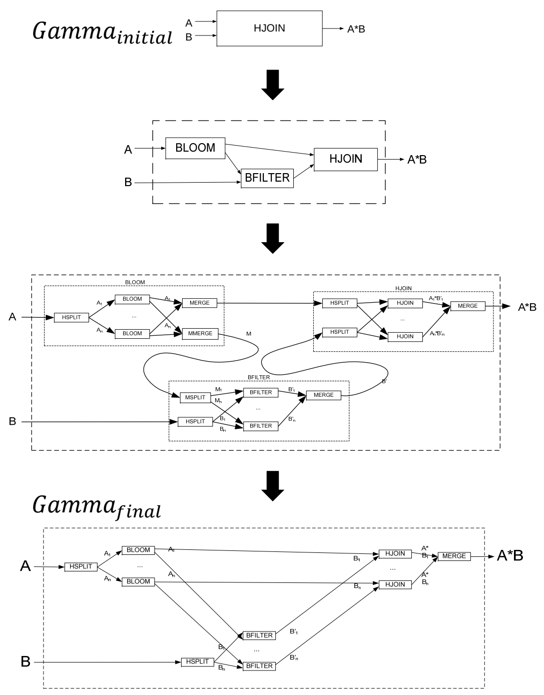


Fig. 8. Derivation of the Gamma Join Algorithm

Why is this so? The primary reason is *performance*. Applications that make DLA kernel calls are common to scientific computing, *eg* simulation of airflow, climate change, and weather forecasting. These applications are run on extraordinarily expensive machines. Time on these machines costs money; higher performance means quicker/cheaper runs or more accurate results. Bottom line: Application developers want the best performance to justify their costs [35].

Consider distributed-memory DLA kernels. They deal with *Single Program, Multiple Data (SPMD)* hardware architectures: the same program is run on each processor, but with different inputs and processors communicate with one another. The operations that a DLA kernel is expected to support is fixed—they have been well-known and well-defined for 40 years. Fig. 9 lists some of the *Level 3 Basic Linear Algebra Subprograms (BLAS3)*,

[7] This is no surprise to scientists. Physics students, for example, typically rederive equations to understand a paper. Similar activities occur in Computer Science.

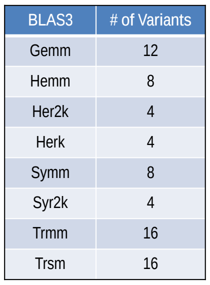
which are matrix-matrix operations [13]. (Level 2 deals with vector-matrix operations and Level 1 vector-vector operations.) There is `Gemm`, general matrix-matrix multiply, Hermitian `Hemm`, symmetric `Symm`, and triangular `Trmm` matrix-matrix multiplies. `Trsm` solves non-singular triangular system of equations.

What is unusual from an SE perspective is that each operation has many variants. Consider `Gemm`. With constants α, β, the general form of this operation is:

$$\mathtt{C} := \alpha \cdot \mathtt{A} \cdot \mathtt{B} + \beta \cdot \mathtt{C}$$

where matrices `A` and `B` are either "normal" or transposed. That's 4 possibilities. Further, the implementation of `Gemm` is specialized for distributed memory based on whether `A`, `B`, or `C` is largest. That's another 3 for a total of $4 \times 3 = 12$. A similar variety is required for other operations.

We also must consider "LAPACK-level" algorithms, which call DLA and BLAS3 operations, such as solvers, factorizations (*eg* Cholesky), and eigenvalue decompositions [1]. We have to generate high-performance algorithms for these operations, too.



BLAS3	# of Variants
Gemm	12
Hemm	8
Her2k	4
Herk	4
Symm	8
Syr2k	4
Trmm	16
Trsm	16

Fig. 9. The BLAS3

Let me be clear: *our work on DLA kernels did not start from scratch*. We mechanized portions of van de Geijn's FLAME project [26] and the distributed-memory DLA library Elemental [42]. FLAME and Elemental leverage 15 years of polishing elegantly layered designs of DLA libraries and their computations. FLAME and Elemental provided the foundation for us to convert dark knowledge of DLA into white knowledge.

4.1 Performance Results

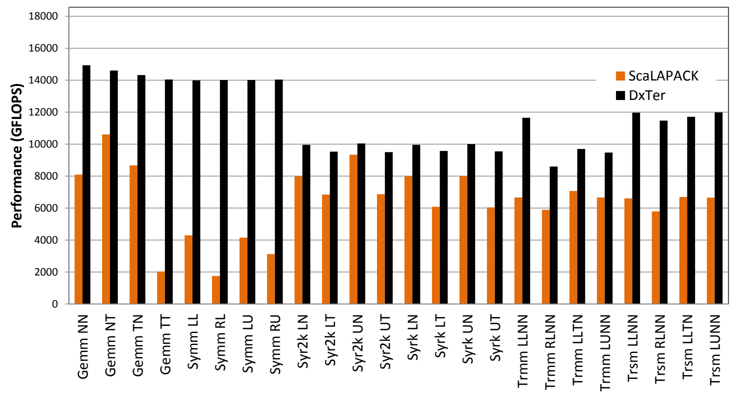
We used two machines in our benchmarks: Intrepid, Argonne's BlueGene/P with 8,192 cores and 27+ TFLOPS peak performance and Lonestar of the Texas Advanced Computing Center with 240 cores and 3.2 TFLOPS peak performance. We compared our results against ScaLAPACK [7], which is the standard linear algebra library for distributed memory machines. Each installation of ScaLAPACK is auto-tuned or manually-tuned. ScaLAPACK was the only DLA library, other than Elemental, for these machines.

DxTer is our tool that generates Elemental code [34,37]. It takes a PIM g of a *sequential* DLA program as input. It exhaustively applies all of the productions $\mathcal{P}$ in its library to produce the space of all of g's implementations $\mathcal{L}((\mathrm{g}, \mathcal{P}))$ *in distributed memory*. Using cost functions to estimate the performance of each derived graph, the most efficient graph is chosen.[8]

We used DxTer to automatically generate and optimize Elemental code for BLAS3 and Cholesky FLAME algorithms. Fig. 10 shows the performance for BLAS3. Overall, DxTer-generated code executes significantly faster than its hand-written ScaLAPACK counterparts. Fig. 11 shows the performance of Cholesky factorization. Again, DxTer generated-code is noticeably faster, which is the same or better than hand-coded Elemental implementations. These graphs are typical of DxTer results [36,37,38].

Today, *Elemental is shipped with DxTer-generated algorithms* [16].

[8] This process of mapping an abstract specification to an efficient implementation is historically called *automatic programming*.



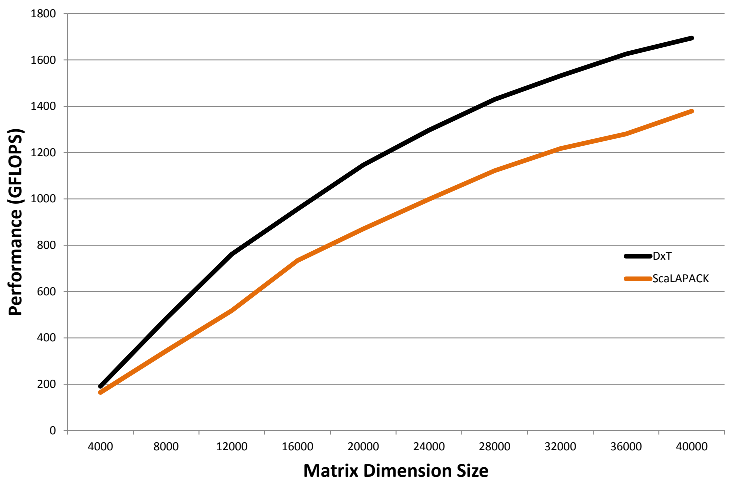


Fig. 11. Cholesky Performance on Lonestar.

4.2 State-of-the-Art vs. Our Group's Vision

Today's linear algebra libraries exist as code. They are rewritten manually as the architecture du jour changes, and it changes rapidly. Consequently, library development lags far behind architecture development, by as much as a decade. Attaining sub-optimal performance on the latest and greatest machines carries a big price for end users.

This is not sustainable. We argue that linear algebra libraries written in a specific language for a specific architecture should never be developed entirely *manually*. Instead, tools, techniques, and theories are needed to encode algorithms, expert knowledge, and information about target architectures. The majority of code libraries can then be generated *automatically*. Experts can overlook optimizations and make mistakes. Machines can not. Performance of generated code is as good as or better than hand-written [36,37,38]. For algorithms that cannot be automatically generated, experts will now have more free time to code them manually. This code can eventually be cast in terms of transformations to encode its dark knowledge. In short, automation will ultimately be a faster, cheaper, better, and more sustainable solution the development of libraries for linear algebra (*cf* [4]).

4.3 DxT Limitations and Salute to Prior Work

DxT is not limited to stateless computations; DxT was originally developed to explain the stateful design of Upright. DxT can be applied to any dataflow domain where the mechanization of rote and/or high-performance code is needed. There are huge numbers of such domains representing great diversity [53].

We would be remiss in not acknowledging prior projects with similar goals and ideas. Among them are the pioneering works on correct-by-construction and deductive program synthesis [24], Amphion [32], rule-based relational query optimization (from which DxT is a direct descendant) [31], SPIRAL [43], the Tensor Contraction Engine [5], and Build-To-Order BLAS [51]. These projects were successful, because their authors put in the effort to make them succeed. Unfortunately, *the successes of these projects are known to far too few in the SE community*. I return to this point in Section 6.

5 Technical Details

With the big picture made clear, let's now drill down to see some details—what in MDE is called the metamodel—of DxT. There are three basic "objects" in DxT: interfaces, primitives, and algorithms. An *interface* is exactly what it suggests: It is a box that defines only the input/output ports and—at least informally—box semantics. A *primitive* is a box that implements a fundamental computation (operation) in code. An *algorithm* is a dataflow graph that references interfaces and primitives.

DxT has two basic "relationships": refinements and abstractions. A *refinement* replaces an interface with an implementation (primitive or algorithm). An *abstraction* rewrites in the opposite direction: from primitive or algorithm to an interface.[9]

Interfaces have preconditions (no surprise). But primitives and algorithms may have preconditions of their own that are *stronger* than the interfaces they implement (this is different). Fig. 12 is a classical example. The `sort` interface takes a stream of records

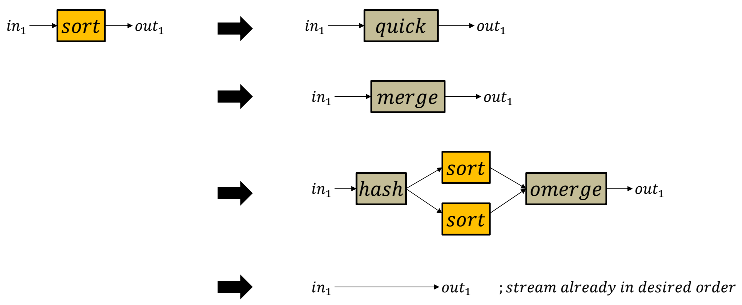


Fig. 12. The `sort` Interface and its Implementations.

as input and produces a sorted stream as output. (The sort key parameter is not shown). The first two refinements show `quick`-sort and `merge`-sort as primitives. The third shows a map-reduce implementation, where `hash` and `omerge` are primitive and any

[9] There is more to DxT, but this is sufficient for this paper. See [21,46] for more details.

implementation of `sort` can be substituted for `sort` interfaces. The last refinement is the focus of this discussion: it says if the input stream is already in sort-key order, nothing needs to be done. This `donothing` algorithm has a precondition that is *stronger* than its `sort` interface.

The *Liskov Substitution Principle (LSP)* is a hallmark of object-orientation [30]. It says if `S` is a subtype of `T`, then objects of type `S` can be substituted for objects of type `T` without altering program correctness. Substituting an interface with an implementing object (component) is standard fare today and is a way to realize refinement in LSP [39,58]. The key constraints of LSP are that preconditions for using `S` can *not* be stronger than preconditions for `T`, and postconditions for `S` are *not* weaker than that for `T`.

The `donothing` refinement is incompatible with LSP. In fact, LSP is too restrictive for graph rewriting; another principle is at work. In 1987, Perry [41] said a box `A` (read: algorithm or primitive) is upward compatible with box `I` (read: interface) iff:

$$\begin{aligned} \mathtt{pre(A)} &\Rightarrow \mathtt{pre(I)} && \textit{; preconditions can be stronger} \\ \mathtt{post(A)} &\Rightarrow \mathtt{post(I)} && \textit{; postconditions can't be weaker} \end{aligned}$$

This is exactly what we need, which we call the *Perry Substitution Principle (PSP)* [41]. It is a practical alternative to LSP that dominates the DxT world. PSP takes into account the local conditions surrounding an interface to qualify legal refinements. *We could not re-engineer legacy designs without it.*

Abstraction—which replaces an implementation with an interface—has stronger constraints than refinement. It implies that a graph `A` must implement `I`. For this to hold, the pre- and postconditions of `A` and `I` must be equivalent [21]:

$$\begin{aligned} \mathtt{pre(I)} &\Leftrightarrow \mathtt{pre(A)} \\ \mathtt{post(I)} &\Leftrightarrow \mathtt{post(A)} \end{aligned}$$

5.1 Optimizations

Earlier we used rewrites that replace a graph α_1 (more than a single node) with another graph α_2 (Fig. 13a), where Fig. 4d is an example. We call these rewrites *optimizations*. Optimizations effectively break "modular" boundaries of adjacent algorithms to expose inefficient graphs which are replaced with more efficient graphs. Optimization is equiv-

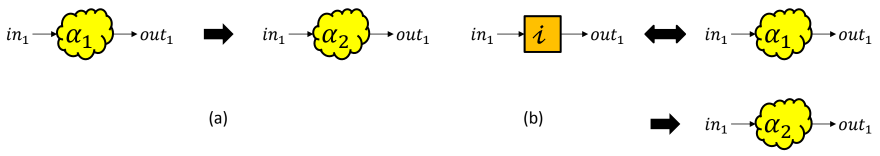


Fig. 13. Optimizing Rewrite Rules.

alent to an abstraction (replacing graph α_1 with interface ι) followed by a refinement (replacing ι with graph α_2) (Fig. 13b). This allows DxT rewrites to assume the canonical form of interfaces on the left and implementations on the right (Fig. 13b).

Optimizations are easily expressed in *n*D grammars. Not so for 1D grammars. Consider the following 1D grammar, where uppercase names are interfaces and lowercase names are primitives:

```
A : a B c | ... ;
B : b | ... ;
```

A sentence of this grammar is `abc`. Suppose composition `bc` implements interface `Z`:

```
Z : bc | q ;
```

Further, domain experts know that `bc` is inefficient and can be replaced by box `q`, which is faster. This is accomplished by abstracting sentence `abc` to `aZ` and then refining to a faster program by replacing `Z` with `q` to yield `aq`. Although this makes perfect sense, abstraction is foreign to 1D grammars and parsing technology [45]; it is a natural part of *n*D grammars. I mentioned earlier (Section 2) that there is an important distinction between 1D and *n*D grammars; this is the point that I wanted to make.

Much of this should look familiar: it is similar to optimization techniques in compilers (esp. for functional languages) [27]. Optimizations break encapsulation boundaries to produce more efficient code. The novelties of DxT are (1) DxT uses graphs not trees, (2) DxT grammars derive software designs, which are *not* parse-trees or ASTs of programs and are *not* states of program executions, and (3) DxT rewrites should be clearly visible to domain experts and program designers, not hidden compiler internals.

5.2 Abstract Interpretation

Another fundamental idea (stolen from compiler playbooks) is *abstract interpretation*. A DxT graph `g` may have many interpretations. The default—what we have used up to now—is to interpret each box of `g` as the computation it represents. `sort` means "sort the input stream". We call this the *standard interpretation* $\mathcal{S}$. The $\mathcal{S}$ interpretation of box `b` is denoted $\mathcal{S}(\texttt{b})$ or simply `b`, *eg* $\mathcal{S}(\texttt{sort})$ is "sort the input stream". The standard interpretation of graph `g` is $\mathcal{S}(\texttt{g})$ or simply `g`.

There are other interpretations. $\mathcal{COST}$ interprets each box `b` as a computation that *estimates the execution time* of $\mathcal{S}(\texttt{b})$ given statistics about $\mathcal{S}(\texttt{b})$'s inputs. So $\mathcal{COST}(\texttt{sort})$ is "return an estimate of the execution time to produce `sort`'s output stream". Each box $\texttt{b} \in \texttt{G}$ has exactly the same ports as $\mathcal{COST}(\texttt{b}) \in \mathcal{COST}(\texttt{G})$, but the meaning of each box and its ports are different.

- We mentioned in Section 4.1 that DxTer forward-engineers (derives) all possible PSMs from an input PIM. The estimated run-time of a PSM `p` is determined by executing $\mathcal{COST}(\texttt{p})$. The most efficient PSM that implements the PIM is the one with the lowest estimated cost [38].
- $\mathcal{M}2\mathcal{T}(\texttt{p})$ is a model-to-text interpretation that maps `p` to executable code.
- Pre- and postconditions help guarantee the correctness of DxT graphs. The $\mathcal{POST}$ interpretation computes properties that are output by a box given properties about box inputs. The $\mathcal{PRE}$ interpretation reads properties about box inputs (computed by $\mathcal{POST}$) and checks if the preconditions of that box are satisfied. A composition of these interpretations ($\mathcal{PRE} \cdot \mathcal{POST}(\texttt{P})$) computes postconditions and validates preconditions of `P` [21].

6 The Reaction to DxT

Fellisen once remarked "It is not a problem to keep ourselves as researchers busy; we need to keep undergraduates busy" [18]. I saved the most important message about DxT for last. DxT is simple enough to teach to undergraduates.

Our thought has always been: once you have a proof or derivation of a design, you've hit the jackpot: you have turned dark knowledge into white knowledge. Having said this, we have been surprised at the reaction to DxT. Some of the reviews we received had breathtaking statements of technical acuity. In Letterman Show countdown order, our top 3 statements are:

3. "Refinement is not a transformation."
2. "Why will you succeed where others have not?"[10,11]
1. "The work lacks motivation."

We were comforted by the fact that conferences were being created solely for rejecting our papers. Overall, none of the reactions made sense to us.

This lead us to conduct user studies involving third-year CS undergraduates and first-year graduates. We split each set of students into two groups. To the first, we presented the big-bang design for Gamma (Fig. 8); to the second, we presented Gamma's derivation. We gave a written quiz that we graded. The result: *no difference!* There was no difference in the number of correct answers, no obvious benefit to DxT derivations over a big-bang. Both undergraduates and graduates were consistent on this point. This was counter-intuitive to us; it didn't make sense.

Perhaps, we thought, Gamma was too simple. So we re-ran the experiment using Upright. The result: again no difference! We were mystified.

Then it occurred to us: maybe these students and referees had *no* experience developing software in this manner. It could not be the case that DxT was difficult to grasp—the ideas are simple. And perhaps also the students and referees had no domain knowledge of parallel relational query processing or crash fault tolerant servers. They could not appreciate what we were telling them. If so, they could not recognize the value on our part to distill software-architecture knowledge as elementary graph rewrites. Nor could they see the practical implication of our results.

We had anecdotal evidence for this last conjecture. We asked ourselves "what are the refinements of `DGemm` (Distributed `Gemm`)?" Of course, we knew the answer (see Fig. 14), but how many others would? People who were familiar with distributed DLA algorithms should know. But very few would know a majority of the rules that we used in DxTer to derive DLA algorithms and how these rules could be applied. In short, the answer was: *very, very few*.

This again brings us back to the differences between 1D and *n*D grammars. It is relatively easy to understand 1D productions—there is little to know. Graph grammars as we use them are different. One needs deep knowledge of a domain to appreciate most rewrites. Very few have such knowledge. Cordell Green once told me "It takes effort" [23]. Few people have been in his (our) position to appreciate this point.

[10] They conveniently ignored our performance results.

[11] Others *have* been successful (Section 4.3). It helps to know the literature.

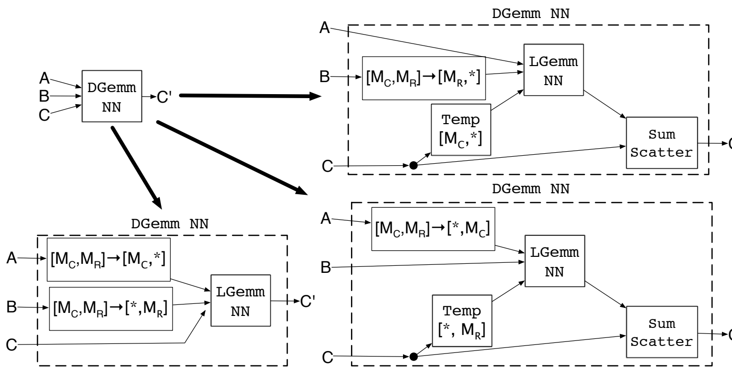


Fig. 14. Distributed Gemm Refinements.

Our next user study in Fall 2012 explored these conjectures. We gave a programming assignment to an undergraduate class of 28 students. We had them build Gamma given its derivation. Once they completed the task, we gave them a questionnaire asking for them to compare their experiences with a big-bang approach (where derivation details were absent). As students had been exposed to big-bang implementations in other classes (and in previous assignments), they could compare DxT with a big-bang. We briefly review some of our questions and results [17,50]:

– **Comprehension**. Do you think the structured way DxT imposes gives you a deeper understanding of Gamma's design than you would get by not using it and doing it your own way?
– **Modification**. Do you think it would be easier or more difficult to modify Gamma with DxT compared to a big-bang approach?
– **Recommendation**. Would you recommend to your fellow students implementing Gamma using DxT or in a big-bang manner?

Analyzing the responses showed that 55% said DxT provided a deeper comprehension of Gamma's design; over 80% said DxT improved comprehension. 47% said it would be considerably easier to modify Gamma given its derivation; over 90% said it would make modification easier. None said it would make it harder. And 88% said they would recommend DxT over a big-bang.

More gratifying were the written comments, a few from different individuals are listed below:

– I have learned the most from this project than any other CS project I have ever done.
– I even made my OS group do a DxT implementation on the last 2 projects due to my experience implementing Gamma.
– Honestly, I don't believe that software engineers ever have a source (to provide a DxT explanation) in real life. If there was such a thing we would lose our jobs, because there is an explanation which even a monkey can implement.
– It's so much easier to implement (using DxT). The big-bang makes it easy to make so many errors, because you can't test each section separately. DxT might take a bit longer, but saves you so much time debugging, and is a more natural way to build things. You won't get lost in your design trying to do too many things at once.

In retrospect, these comments were familiar. In October 2003, NSF held a Science of Design Workshop in Airlie, Virginia. Fred Brooks (1999 Turing Award) summarized the conclusions of his working group to explore the role of science in design: "We don't know what we're doing and we don't know what we've done!". To paraphrase Edsger Dijkstra (1972 Turing Award): "Have you noticed that there are child prodigies in mathematics, music, and gymnastics, but none in human surgery?". The point being that there are bodies of knowledge that take years to comprehend and there are no short-cuts to achieve such understanding. We owe our success with DxTer to 15 years of research by van de Geijn and others to understand the domain of DLA. Not all domains are this hard to understand, but again, *it takes effort*. Our take-away conclusion is this:

> *Knowledge, experience, and understanding how to codify knowledge of efficient programs in a reproducible way is everything to automated design. Lacking any of these is a significant barrier to progress.*

7 Conclusions

Programmers are geniuses at making the simplest things look complicated; finding the underlying simplicity is the challenge. Programmers are also geniuses at making critical white knowledge dark; reversing the color of knowledge is yet another challenge. It takes effort to understand a legacy application or domain to mine out its fundamental identities or rewrite rules that are key to (a) automated design, (b) correct-by-construction, and (c) transforming undergraduate education on software design from hacking to a more scientific foundation.

Software Language and Engineering (SLE) has great potential for the future of Software Engineering. Formal languages will be the foundation for automated software development. Knowledge of dataflow application designs will be encoded as graph grammars, not Chomsky string grammars, whose sentences define complex programs. Such grammars will enable the design of programs to be optimized automatically; they will remove the burden of rote, tedious, difficult, and error-prone activities of program development; they will scale domain expertise from a few people to the masses; and most importantly, they ultimately will help modernize undergraduate curriculums in software design.

Acknowledgements. We thank R. van de Geijn (Texas), T. Riche (NI), M. Erwig (Oregon), R. Paige (York), and C. Kästner (CMU) for their helpful comments on drafts of this paper. We gratefully acknowledge support for this work by NSF grants CCF-0724979, CCF-0917167, and ACI-1148125. Gonçalves was funded by the ERDF project FCOMP-01-0124-FEDER-010152 and FCT grant SFRH/BD/47800/2008. Marker held fellowships from Sandia National Laboratories and NSF (grant DGE-1110007). Siegmund was funded by BMBF project 01IM10002B. This research used resources of the Argonne Leadership Computing Facility at Argonne National Lab, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. We are greatly indebted to Jack Poulson for his help to understand his Elemental library.

References

1. Anderson, E., et al.: LAPACK Users' Guide. SIAM, Philadelphia (1992)
2. Bahcall, J., Piran, T., Weinberg, S.: Dark matter in the universe. In: 4TH Jerusalem Winter School For Theoretical Physics (1987)
3. Batory, D., Azanza, M., Saraiva, J.: The Objects and Arrows of Computational Design. In: MODELS (2008)
4. Batory, D., Singhal, V., Sirkin, M., Thomas, J.A.: Scalable software libraries. In: SIGSOFT (1993)
5. Baumgartner, G., et al.: Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. Proceedings of the IEEE (2005)
6. Baxter, I.D.: Design Maintenance Systems. CACM (April 1992)
7. Blackford, L.S., et al.: ScaLAPACK: a portable linear algebra library for distributed memory computers - design issues and performance. In: SC (1996)
8. Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., Riche, T.: Upright cluster services. In: SOSP (2009)
9. Curtis, B., Krasner, H., Iscoe, N.: A field study of the software design process for large systems. Comm. ACM (Nov 1988)
10. D'Antonio, F.: http://www.docstoc.com/docs/123006845/Introduction-to-Graph-Grammars-DAntonio (Oct 2003)
11. Derk, M., DeBrunner, L.: Reconfiguration graph grammar for massively parallel, fault tolerant computers. In: Graph Grammars and Their Application to Computer Science, vol. 1073. Springer Berlin Heidelberg (1996)
12. Dewitt, D.J., Ghandeharizadeh, S., Schneider, D., Hsiao, A.B.H., Rasmussen, R.: The Gamma Database Machine Project. IEEE ToKaDE 2(1) (1990)
13. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Software 16(1) (Mar 1990)
14. Dowling, J., Cahill, V.: Dynamic software evolution and the k-component model. In: Workshop on Software Evolution at OOPSLA (2001)
15. Ehrig, H., Pfender, M., Schneider, H.J.: Graph-grammars: An algebraic approach. In: SWAT (1973)
16. Elemental Team. http://libelemental.org/about/team.html
17. Feigenspan, J., Batory, D., Riché, T.L.: Is the derivation of a model easier to understand than the model itself? In: ICPC (2012)
18. Felleisen, M.: Private Correspondence (Jan 2007)
19. Ferrucci, F., Tortora, G., Tucci, M., Vitiello, G.: A predictive parser for visual languages specified by relation grammars. In: VL (1994)
20. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In: MODELS (2006)
21. Gonçalves, R.C., Batory, D., Sobral, J.: ReFlO: An interactive tool for pipe-and-filter domain specification and program generation. submitted (2013)
22. Green, C., Luckham, D., Balzer, R., Cheatham, T., Rich, C.: Report on a knowledge-based software assistant. Tech. rep., Kestrel Institute (1983)
23. Green, C.: Private Correspondence (Jan 2009)
24. Green, C., Luckham, D., Balzer, R., Cheatham, T., Rich, C.: Report on a knowledge-based software assistant. Kestrel Institute Technical Report KES.U.83.2 (1983)
25. Grunske, L., Geiger, L., Zündorf, A., Van Eetvelde, N., Van Gorp, P., Varro, D.: Using graph transformation for practical model driven software engineering. In: Model-Driven Software Development. Springer Berlin Heidelberg (2005)

26. Gunnels, J.A., Gustavson, F.G., Henry, G.M., van de Geijn, R.A.: FLAME: Formal Linear Algebra Methods Environment. ACM Trans. on Math. Softw. (Dec 2001)
27. Jones, S.L.P., Santos, A.L.M.: A transformation-based optimiser for haskell. Science of Computer Programming 32(1–3) (1998)
28. Königs, A., Schürr, A.: Tool integration with triple graph grammars - a survey. Electronic Notes in Theoretical Computer Science 148(1) (2006)
29. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. 16(2) (1998)
30. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. 16(6) (1994)
31. Lohman, G.M.: Grammar-like functional rules for representing query optimization alternatives. In: ACM SIGMOD (1988)
32. Lowry, M., Philpot, A., Pressburger, T., Underwood, I.: Amphion: Automatic programming for scientific subroutine libraries. In: ISMIS (1994)
33. Maggiolo-Schettini, A., Peron, A.: A graph rewriting framework for statecharts semantics. In: Graph Grammars and Their Application to Computer Science, vol. 1073. Springer Berlin Heidelberg (1996)
34. Marker, B., Batory, D., Shepherd, C.: Dxter: A dense linear algebra program synthesizer. Computer Science report TR-12-17, Univ. of Texas at Austin (2012)
35. Marker, B., Batory, D., van de Geijn, R.: DSLs, DLA, DxT, and MDE in CSE. In: SECSE (May 2013)
36. Marker, B., Batory, D., van de Geijn, R.: A case study in mechanically deriving dense linear algebra code. International Journal of High Performance Computing Applications (To Appear)
37. Marker, B., Batory, D.S., van de Geijn, R.A.: Code generation and optimization of distributed-memory dense linear algebra kernels. In: ICCS (2013)
38. Marker, B., Poulson, J., Batory, D.S., van de Geijn, R.A.: Designing linear algebra algorithms by transformation: Mechanizing the expert developer. In: VECPAR (2012)
39. Medvidovic, N., Rosenblum, D.S., Taylor, R.N.: A language and environment for architecture-based software development and evolution. In: ICSE (1999)
40. Müller, H.: Private Correspondence (May 2013)
41. Perry, D.E.: Version control in the inscape environment. In: ICSE (1987)
42. Poulson, J., Marker, B., van de Geijn, R.A., Hammond, J.R., Romero, N.A.: Elemental: A new framework for distributed memory dense matrix computations. ACM Trans. on Math. Softw. 39(2) (Feb 2013)
43. Püschel, M., et al.: SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE, special issue on “Program Generation, Optimization, and Adaptation” (2005)
44. Rekers, J., Schürr, A.: Defining and parsing visual languages with layered graph grammars. Journal of Visual Languages & Computing 8(1) (1997)
45. Rich, E.A.: Automata, Computability and Complexity: Theory and Applications. Pearson-Prentice Hall (2008)
46. Riché, T., Goncalves, R., Marker, B., Batory, D.: Pushouts in Software Architecture Design. In: GPCE (2012)
47. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation, Vol I: Foundations. World Scientific (1997)
48. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. (2009)
49. Schürr, A.: Introduction to progress, an attribute graph grammar based specification language. In: Graph-Theoretic Concepts in Computer Science. Lecture Notes in Computer Science, vol. 411. Springer Berlin Heidelberg (1990)
50. Siegmund, J.: Framework for Measuring Program Comprehension. Ph.D. thesis, University of Magdeburg, School of Computer Science (2012)

51. Siek, J.G., Karlin, I., Jessup, E.R.: Build to order linear algebra kernels. Parallel and Distributed Processing (2008)
52. Taentzer, G.: Agg: A graph transformation environment for modeling and validation of software. In: Applications of Graph Transformations with Industrial Relevance, vol. 3062. Springer Berlin Heidelberg (2004)
53. The LabVIEW Environment. `http://www.ni.com/labview/`
54. Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt: A language for streaming applications. In: Conference on Compiler Construction (2002)
55. Tichy, M., Henkler, S., Holtmann, J., Oberthür, S.: Component story diagrams: A transformation language for component structures in mechatronic systems. In: Workshop on Object-oriented Modeling of Embedded Real-Time Systems, Paderborn, Germany (2008)
56. Wermelinger, M., Fiadeiro, J.L.: A graph transformation approach to software architecture reconfiguration. Sci. Comput. Program. (2002)
57. Wikipedia: Graph rewriting. `http://en.wikipedia.org/wiki/Graph_rewriting`
58. Wikipedia: Component-based software engineering. `http://en.wikipedia.org/wiki/Component-based_software_engineering` (2013)
59. Wittenburg, K.: Earley-style parsing for relational grammars. In: Visual Languages (1992)