Intelligent Components and Software Generators[1]

Don Batory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
batory@cs.utexas.edu

The production of well-understood software will eventually be the responsibility of software generators. Generators will enable high-performance, customized software systems and subsystems to be assembled quickly and cheaply from component libraries. These components will be intelligent: they will encapsulate domain-specific knowledge (e.g., "best practice" approaches) so that their instances will automatically customize and optimize themselves to the system in which they are being used. In this paper, we explore the topics intelligent components and software generation as they pertain to the issues of software productivity, performance, reliability, and quality.

1 Introduction

It is commonplace to create customized PC hardware configurations through the assembly of plug-and-play hardware components. In the future, customized software systems will be assembled in much the same manner using plug-and-play software components. The availability of plug-and-play software technologies will mark a major advance in software construction. Components will offer substantial increases in software productivity, performance, reliability, and quality. The increase in productivity is easy to understand: components are reused, not rewritten. They will be written by domain experts so that the quality of component software is substantially higher than that which can be produced by typical programmers. The reliability of components will exceed that of typical software modules because they will have been tested in a wider variety of situations. Finally, components will encapsulate "best practice" approaches for coding and run-time tuning: rather than expecting application performance to be degraded, the expectation will be that application performance will *improve* with the use of components.

Components are available today through platforms like CORBA and Microsoft OLE and COM [Ude94]. These technologies (and their variants) simplify the task of *manually* integrating components to build applications. A more advanced view is the use of generators that *automatically* assemble applications from specifications of component compositions. It is this advanced view of software development that we are realizing at our research lab at the University of Texas. We have developed tools that glue pre-written components together to assemble high-performance, customized software systems in minutes [Bat92-93, Bat97]. The components that we use encapsulate and automate domain-specific techniques that are best-practice for building software, so that generated code is efficient. We have developed tools that allow us to validate the correctness of our compositions automatically, so that we know at system specification time whether or not the system that we have designed will indeed work [Bat96]. We are now developing tools to critique our designs automatically, given a workload specification. These tools (or *design wizards*) will suggest alternative component compositions to those that we have created manually, and will provide reasons why the suggested design changes will improve the overall product.

1. This research is supported by DARPA contract F30602-96-2-0226, Microsoft Research, and The University of Texas Applied Research Laboratories.

Software architectures has emerged over the last four years as a significant area of study in software engineering [Per92, Gar95]. The basic goals are (1) to understand the problems of building software from components, (2) to find solutions to these problems, and (3) to evaluate trade-offs to determine which solution to use if multiple solutions can be applied. A subarea of software architectures is *software generators* [Bat94b, Kie96]. Generators take component assemblies as input specifications and produce optimized source code as their output. The basic distinction between research on generators and software architectures is that the components that generators compose to construct systems are designed to be plug-compatible, interchangeable, and interoperable. These assumptions are not common for general work on software architectures. However, the advantages of making these assumptions are substantial and their consequences are the topic of this paper.[2]

At the core of generator technology is software reuse. To appreciate the problems in reuse that generator technologies have overcome, it is instructive to review results of Selby [Sel88]. In 1988, Selby examined the cost of software reuse in Fortran libraries. In particular, he posed the question: "What is the cost of reusing a module if n% of it were modified?", where n=0 means "as is" reuse, and n=100 means coding a module from scratch. Figure 1 displays his findings. Note that "as is" reuse is not free: there is a small (but noticeable) cost for reuse, because one must invest the time to understand what a component does before deciding that it can be used. If minor modifications are needed, denoted in his studies by n<25%, there is a substantial penalty — 55% of the cost of rewriting the module from scratch. The reason is simple: one must have deep knowledge of how a module is implemented before it can be modified correctly, and acquiring this knowledge takes substantial time. Interestingly, the cost of a major modification (n≥25%) is marginally more expensive (70%). These figures are typical to that which we faced in building reusable components of generators.

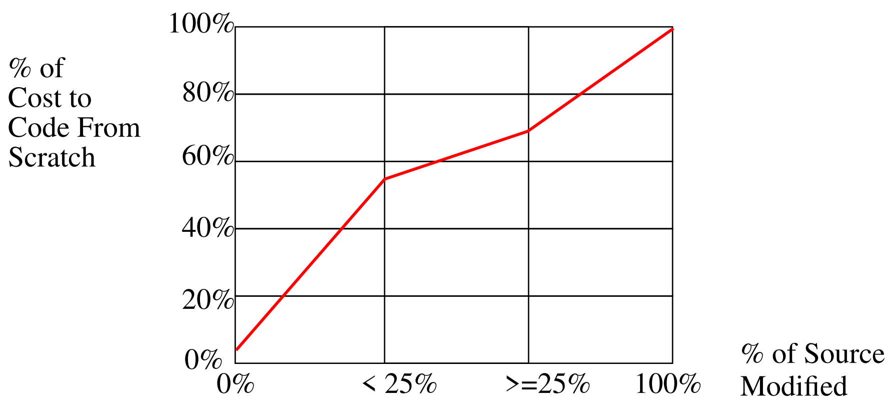

Figure 1: Costs of Software Reuse [Sel88]

We learned two major lessons about component reuse in building generators. The first is: *don't manually modify components*. By doing so, one should expect (as Selby's results indicate) a major loss in productivity. Clearly, what we want is "as is" or black box reuse. The second lesson is: *component modification is unavoidable*. When a component is used in a new setting, it must often be modified for performance or compatibility reasons. Rather than modifying components manually, we encoded domain-specific intelligence into components so that they perform the customizations and optimizations automatically — tasks that are now performed manually by domain experts. This is the idea of *intelligent components*.

2. Actually, very few software generators are based on component technologies. Common compiler-generator tools like **`lex`** and **`yacc`**, for example, do not rely on software components. It is this newer (more uncommon) class of generators that is the subject of this paper.

Intelligent components are the central idea and strength of generators. They have effectively altered the shape of Selby's cost-of-reuse-curve: the curve representing the cost of reuse in generators is a flat line that hugs the X-axis. Intelligent components make "as is" reuse practical. They have allowed us to achieve significant productivity gains while retaining high quality and reliable software without loss in application performance.

Much of this sounds too good to be true. It is indeed the case that generators and their component technologies are not free. There are at least three qualifications that must be satisfied for generator technologies to be applicable. First, the domain in which software is to be generated must be mature. The problems of software construction must be well-understood, and so too their solutions. Component technologies are the product of standardizing these problems and their solutions. Second, it must be economical; generators are expensive to build. Thus, it should be common in the domain for there to be many variants of a single system, and that building each variant is expensive. Third, it must be feasible, both technically and politically. From our experience, if a domain satisfies the first two qualifications — maturity and economics — we have found the technical problems are solvable. To be sure, there are difficult technical challenges ahead, but there are no show-stoppers. The hard part is political feasibility. Many organizations are not yet prepared to automate the development of software that they are manually producing today. In this paper, we focus on the technical aspects and accomplishments of generators.

In the following sections, we provide background material to give the readers more insights on generators. We showcase the P2 generator as an example to make our points concrete. We then present a spectrum of results from different experiments, to convey the broad capabilities of generators. Our goal is to use these results to demonstrate the improvements generators offer in terms of software productivity, performance, reliability, and quality. We conclude with a brief discussion of our current research.

2 Background

In this section, we review selected concepts on a class of generators called GenVoca generators, and explain the scope of their application. A key point we want to convey is our use of the term "generator" is actually much broader than its conventional meaning. So the ideas and results that we report on in this paper actually have wide applicability.

2.1 GenVoca

GenVoca is a scalable model of constructing software systems from components. It is scalable because exponentially-large families of systems can be constructed with a small number of components [Bat93, Big94]. GenVoca generators have been created for many domains. Most that are listed in Table 1 have been designed, conceived, and built independently of each other. This means that researchers are regularly reinventing the same ideas; because there is a lot of trial-and-error, reinvention has a tremendous cost. One of the important aspects of our research is to distill the experiences of these projects to expose a general methodology and techniques for generator construction so that others can more easily build GenVoca generators for their domains.

The first GenVoca generator (to our knowledge) was Genesis: it demonstrated how customized DBMSs could be assembled from prefabricated components. Its counterpart in the domain of network protocols was Avoca/x-kernel. Complex protocol suites were constructed by composing plug-compatible components called *micro-protocols*. Ficus is GenVoca generator for constructing scalable distributed file systems for Unix. ADAGE is a generator for avionics software; SPS is generator for signal processing. Our most recent work is generators for container data structures (P2 and DiSTiL).

Domain	Generator Name	Year	References
database management systems	Genesis	1988	[Bat88]
network protocols	Avoca/*x*-kernel	1989	[Hut91]
file systems	Ficus	1990	[Hei94]
avionics	ADAGE	1992	[Cog93]
data structures	P2/DiSTiL	1994/1996	[Bat94b, Sma97]
audio signal processing	SPS	1996	[Alt96]

TABLE 1. GenVoca Generators

There are three central ideas that characterize GenVoca generators. First, the building blocks of software systems are *refinements* of fundamental domain abstractions (see also [Nei89, Bax92]). Each refinement corresponds to some feature of a domain that can be shared by many systems. The implementation of a refinement is a *component*; it is through domain engineering (where interface standardizations are imposed) that these components are defined to be plug-compatible, interchangeable, and interoperable [Pri91]. Second, GenVoca refinements are *large-scale*. That is, they define a consistent refinement of multiple classes of a target system. The idea is familiar to software engineers: when a new feature or capability is added to an existing application, modifications to the application's source code are not localized; one must update application source code in multiple places. GenVoca components formalize this concept and automate the modification of target system software when a component is added or removed from a system. Third, GenVoca refinements are *parameterized*. Parameterization is a simple and elegant model of component customization and composition [Gog86-96].

Even at this very high level of description, there are distinctions among different generators beyond their domain of applicability. In particular, generators can be distinguished by how they answer two questions:

- When are refinements composed?
- How are refinements implemented?

Refinements (or components that implement refinements) can be composed either statically or dynamically. *Static* compositions mean that components are composed at application generation time; compositions are fixed for the lifetime of the application. In contrast, *dynamic* compositions mean that components are composed at application run-time. Thus, this enables applications to be dynamically reconfigurable.

Refinements can be implemented in two ways: compositionally or transformationally [Big87, Gri94]. A *compositional* implementation — called a *compositional component* — defines the code that is to be executed at application run-time. Compositional components are templates (if composed statically) or are object libraries (when composed dynamically). A *transformational* implementation — called a *transformational component* — is code that generates the code that an application is to execute at run-time. In effect, transformational components generate compositional components that are specifically optimized and customized to a particular application. Transformational components are most commonly implemented as domain-specific extensions to compilers of general-purpose programming languages. Table 2 shows the classification of the generators listed in Table 1.

	Compositional	Transformational
Static	Genesis, ADAGE, Ficus	P2/DiSTiL
Dynamic	Avoca/*x*-kernel	SPS

TABLE 2. Classification of GenVoca Generators

When building a generator for a domain, one must choose between a compositional or transformational implementation of all of its components. Presently, generator technologies do not permit a mixture of dynamic and static, compositional and transformational components. Rather, all components that a generator composes must be implemented the same way. Thus, this means that the implementors of generators face an important decision very early: they must decide whether or not to use dynamic or static compositions, and whether their components are to have a compositional or transformational implementation.

Presently, we have few guidelines to make this decision wisely. In particular, the tradeoffs of choosing between compositional and transformational implementations are not well understood. On the one hand, compositional implementations are simple to understand: one writes the code that is to be executed by an application. On the other hand, transformational implementations are more complicated, as there is a "level of indirection" one must contend with. Namely, one must write code that *generates* the code that applications are to execute.

Generally speaking, there is adequate infrastructure for writing compositional components, as vanilla programming languages can be used.[3] The problems are more serious for transformational components, because they are *metaprograms* — programs that write other programs [Kic91]. This means that transformational components represent and manipulate programs as data. The LISP and Scheme communities have long recognized and solved these problems: both programs and data are represented as prefix expressions [Kic91, Spr90]. However, for industrial languages like C and C++ (and the newcomer Java), there is no language support for representing and manipulating programs as data. Consequently, a critical aspect of programming infrastructure is lacking.

Finally, we've observed that compositional technologies are appropriate when one is reusing "larger" components (in excess of 10,000 lines each), where relatively few optimizations and customizations are required and performance is not critical. At the other extreme, transformational technologies are appropriate when one is reusing "small" components that each contribute a few (e.g., less than 100) lines of code to an application, where there are many domain-specific optimizations to be performed, and where performance of the generated code is critical. In between these two extremes is a grey area where there are few guidelines to follow for choosing component implementation technologies.

This background material is important to the subject of this paper because our use of the term "generator" is quite broad: it can range from conventional template and object libraries to classical code generators. The key point that we want to stress is that GenVoca is a powerful model. It enables us to understand issues of software architecture in terms of primitive domain-specific refinements and their compositions. It tells us the primitives to use to conceptualize software component technologies, thereby permitting domains and systems to be analyzed in largely implementation-independent ways. This is exactly the kind of features/properties that (we feel) are needed for generators and for understanding software architectures.

3 The P2 Generator

P2 is a static, transformational generator of container data structures. It is static because components are composed at application generation time; it is transformational because its components generate the code that applications are to execute. A version of P2, called DiSTiL [Sma97], has been implemented as part of Microsoft's Intentional Programming (IP) project, a prototype program transformation system that is under

3. This is not strictly true. Dynamic compositions of components often corresponds to the dynamic creation of inheritance hierarchies — i.e., the ability at run-time to alter the superclass(es) of a given subclass. Most programming languages support only statically-defined inheritance hierarchies.

development at Microsoft Research [Sim95]. IP will be used to develop software internally at Microsoft before being released as a product.

It often surprises readers to learn that the study of data structures would have any bearing on understanding issues of software architecture. The typical reaction is: “Aren’t data structures trivial?” “Why even look at this domain at all?” Indeed, compared to avionics, file systems, and network protocols, data structure algorithms are simple. But the focus of software architectures and software componentry is not on algorithm complexity but in understanding the mechanisms by which components are defined, composed, customized, and parameterized. We have found these mechanisms to be the same (or even *more* complex) for data structures than they are for avionics, file systems, etc. Thus, data structures is an ideal domain for study: its algorithms are understood by all and it is easy to illustrate how component technologies work. By showing how a readily understood domain can be decomposed into components, and that complex data structures can be synthesized through component composition, it is more likely that others can understand how to apply these ideas to their own, more complicated, domains.

P2 abstractions are elementary. Typical container data structures, such as binary trees, lists, etc., are implementations of a simple cursor-container abstraction (see Figure 2). A *container* is a collection of *elements* of a single type. A *cursor* is a run-time object that is used to retrieve, update, and delete elements of a container. Users code their P2 programs in terms of cursor and container abstract data types; these types are abstract because they have no implementation. The P2 generator manufactures their implementation at P2-program compile time.

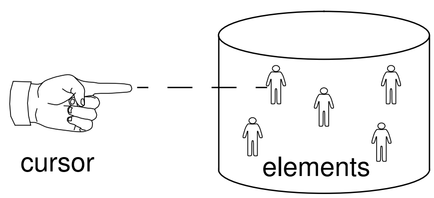

Figure 2: P2 Cursor-Container-Element Abstractions

More specifically, a P2 program is a C program that references abstract cursor and container data types (which P2 added to the C language). At the top of a P2 program is a *type equation*, which specifies a composition of P2 components that defines how cursor and container data types are to be implemented. The P2 program is then submitted to the P2 generator (which really is a C-preprocessor) that outputs the same C program, but now implementations have been generated for the referenced cursor and container types.

P2 gives containers a relational database-like “image”. Not surprisingly, it’s syntax is also relational-like. For example, a container of elements (**emp_cont**) that are instances of the **EMPLOYEE_TYPE** is declared below:

```
container< EMPLOYEE_TYPE > emp_cont;
```

A cursor that ranges over all elements of **emp_cont** is **all_emp**:

```
cursor< emp_cont > all_emp;
```

selected_emp is a cursor that ranges only over those elements of **emp_cont** where the department number is 10:

```
cursor< emp_cont > where "$.deptno == 10" selected_emp;
```

Given the above, a code fragment that retrieves all employees whose department number is 10, prints out the names of these employees, and deletes these employees is easily expressed:

```
foreach (selected_emp) {
   printf("%s\n", selected_emp.name);
   delete(selected_emp);
}
```

While admittedly this code fragment is artificial, the basic idea is clear: programmers are freed from data structure implementation details and can concentrate on the algorithms that are specific to their application.

P2 presently has over 50 components in its library. Among them include components for AVL trees, splay trees, doubly-linked lists, key-ordered doubly-linked lists, compression algorithms (i.e., algorithms that transform containers of uncompressed elements to containers of compressed elements), memory managers, sequential and heap storage of elements, and components that store containers in transient or persistent memory. Components are organized into *realms* — i.e., libraries of plug-compatible and interchangeable components. A partial listing of the membership of the two major realms of P2, **DS** (data structures) and **MEM** (memory storage), is given below:

```
DS  = { avl[DS]         // avl tree
        splay[DS]       // splay tree
        dlist[DS]       // doubly-linked list
        odlist[DS]      // key-ordered doubly-linked list
        compress[DS]    // compression of elements
        avail[DS]       // memory managers
        sequential[MEM] // sequential storage
        heap[MEM]       // heap storage
        ...
      }

MEM = { transient       // transient memory
        persistent      // persistent memory
      }
```

A *type equation* is a named expression that defines a composition of P2 components; it defines how the cursor-container abstraction (Figure 2 and Figure 3a) that users use to code their programs is to be implemented. Suppose we want to store elements of a container using an AVL tree. To do so, we add the P2 **avl** tree component to the container's type equation. Figure 3b shows the result of this refinement: elements have sprouted left and right AVL tree pointers and the container has "grown" a pointer to the root of the AVL tree. (This, incidentally, is an example of a large-scale refinement: both element types and container types are refined as a consequence of using the **avl** component).

Next, suppose we want to store the nodes of the AVL tree in a heap. To do so, we add the P2 **heap** component to the type equation (Figure 3c). The result is that elements now are assigned heap addresses that were not known in Figure 3b.

Lastly, we want to store elements in transient memory. To do so, we insert the **transient** component into the equation (Figure 3d). Now all heap addresses reside in transient memory. Thus the type equation **simple = avl[heap[transient]]** means that we are storing elements of a container in an AVL tree, whose nodes are stored in a heap in transient memory.

Now suppose after fielding the application we discover that it is spending more and more time loading the container with elements each time it is executed. Rather than storing elements in transient memory (requiring them to be reloaded upon each execution of the application), it is more efficient to store the contents of the container in persistent memory, so that all an application needs to do is to open the persistent container.

This can be accomplished by replacing the **transient** component with the **persistent** component (Figure 3e). Now the container is stored in a persistent file "**foo**" (where "**foo**" is a user-supplied parameter to the persistent layer); elements are now assigned persistent (heap) addresses.

Finally, suppose later we discover that we would like to be able to read elements in an order that is different than that provided to us by the AVL tree, yet we still want to retain the use of the AVL tree as a data structure for these elements. This can be accomplished by inserting an ordered-doubly-linked list component **odlist** into the type equation (Figure 3f).[4] Now, elements of our container are stored on an ordered list, they are also interconnected by an AVL tree, whose nodes are stored in a heap in persistent memory.

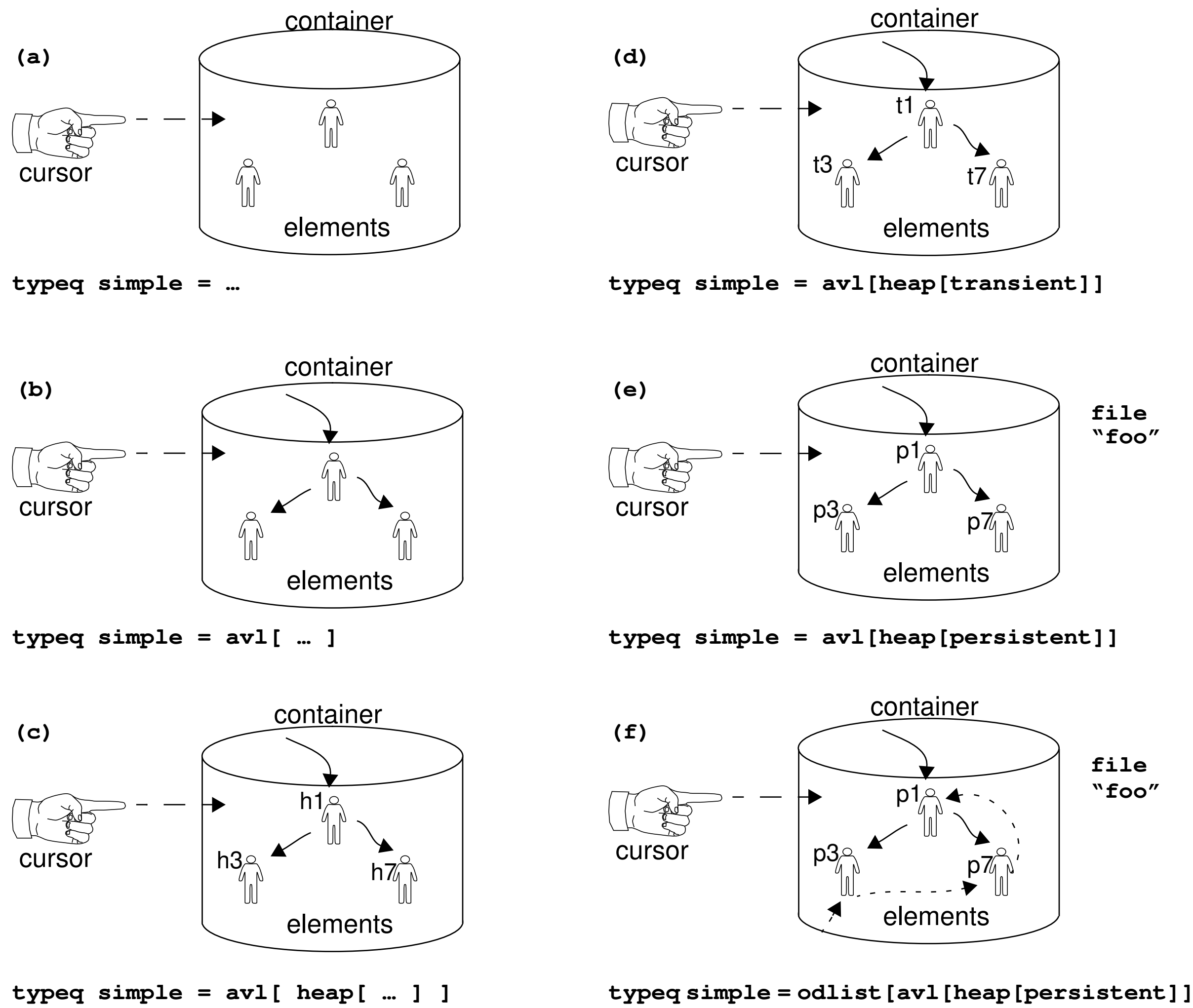

Figure 3: Refinements of a Cursor-Container Abstraction

The point of this example is to illustrate how data structures of considerable complexity are easy to specify and assemble from components. Once readers understand how components "refine" the cursor-container abstractions by grafting on implementation details, and how compositions of components progressively

4. The key field of this list, like the key of the AVL tree, is not explicit in our example, but is explicit in an actual implementation.

reveal these details, it is not difficult to imagine how these same ideas have been applied in the creation of generators for database systems, file systems, avionics, and so on. This is the central idea of GenVoca.

4 Experimental Results

In this section, we review results from selected projects and experiments to paint a general picture of improved software productivity, performance, reliability, and quality.

4.1 LEAPS

LEAPS is a production system compiler [Mir90, Bra93] that produces the fastest executables of OPS5 rule sets [Coo88]. LEAPS translates an OPS5 rule set — a set of rules with actions to be performed when a rule is fired — into a C program. When the C program executes, it fires these rules at a rate which can be an order of magnitude or more faster than OPS5 interpreters.

LEAPS is an interesting application because the C programs that it generates relies on unusual container data structures and search algorithms that are unlikely to be found in any generic data structure library. LEAPS is also a performance-driven application; it was hand-tuned and hand-coded by experts. Finally, it was well-known that the LEAPS algorithms were difficult to understand. Thus, LEAPS seemed to be an "acid test" for P2: it was our belief that if we could do well in reengineering LEAPS using P2, P2 could do well on most any data structure application.

Our P2 re-engineered version of LEAPS, called RL, performed much better than we had imagined. We had a 4-fold reduction in code volume: LEAPS is 20K lines of code; RL was about 5K. P2 (which itself is 50K lines of code) provided us with tremendous leverage. In fact, our productivity in building RL was at the rate at which experts in the domain could code, yet we certainly were novices: we had never dealt with rule systems prior to RL. When we benchmarked the C files produced by LEAPS and RL, we found that, on average, the RL-produced files executed about 50% faster than LEAPS executables [Bat94b]. Upon closer inspection, we discovered that a contributing factor was that P2 could perform optimizations automatically that were difficult, if not impractical, to do by hand. Finally, our P2 specifications of the LEAPS algorithms (expressed in terms of cursors and containers) were very clean and easy to understand [Bat94a]. It revealed the underlying elegance of LEAPS, but also suggested ways in which to improve its performance by swapping/adding P2 components to the type equations that defined the implementations of LEAPS containers.

Figure 4 shows the performance results for a typical rule set, **`waltz`**. The Y-axis indicates run-time in seconds (logarithmic, base 10); the X-axis shows data set size. The top two curves show the performance of two versions of LEAPS: the highest (slowest) is the persistent version called DATEX. (DATEX containers resided in persistent storage that used the relational storage manager of Genesis [Bat88, Bra93]). LEAPS (which used transient containers) is the second highest curve. The performance of RL (for both transient and persistent versions) are the two lines below that of LEAPS. The persistent version of RL was achieved by unplugging the transient component and replacing it with a memory-mapped persistent component. (Because memory mapped I/O only results in a degradation of about 10%, there was little performance difference between transient and persistent versions of RL).

We mentioned earlier that because the P2 specification of LEAPS revealed the source of much of the CPU overhead, we introduced new hashing components/data structures to the type equation that defined RL containers. By doing so, we were able to improve performance further (the bottom-most curves of Figure 4). Our persistent hashed-version of RL ran over 40 times *faster* than LEAPS! [Bat97]

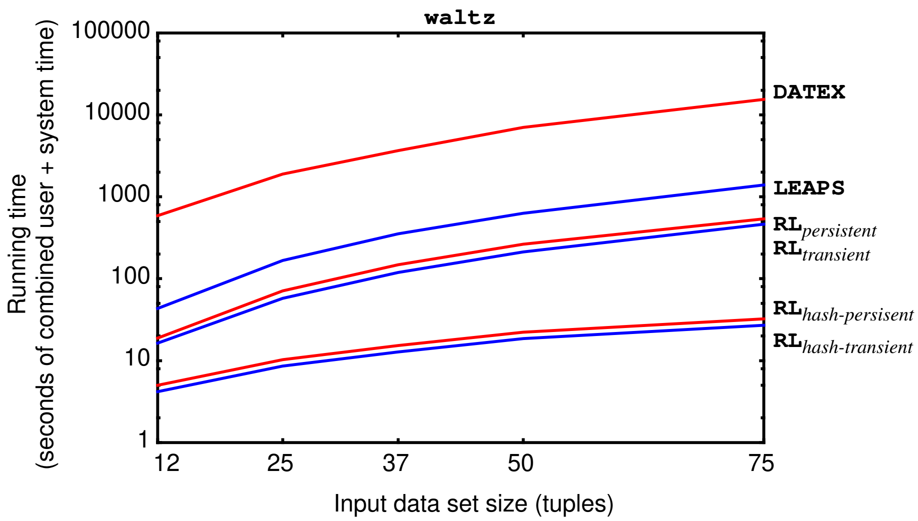

Figure 4: Performance Comparison of RL with LEAPS

We learned an important lesson in software quality from our experiments: abstracting voluminous and complex implementation details promotes clean, efficient, and understandable designs. Moreover, organizing fewer details often leads to more maintainable products. In our case, it was very easy to alter design decisions after we had built RL: we simply redefined type equations (to alter the implementation of its containers) and recompiled; no other source code was changed.

4.2 LRUsim

The *Object-Oriented Programming Systems (OOPS)* Group at the University of Texas is investigating issues of OS memory management. One of their projects is to analyze the locality of references in memory using real program traces. Their observation is that randomly generated memory references do not reflect the true usage of memory; the proper way to evaluate memory management schemes is to use real traces.

LRUsim is a hand-coded C++ tool that maintains the LRU ordering of pages using container data structures. LRUsim reads a trace file and maintains a page reference queue; it records the position of each page in the queue at reference time and promotes that page to the head of the queue. If the memory size is m pages, references to positions $1 \ldots m$ represent page hits; above m are misses (page faults). The idea naturally extends to memory hierarchies: level 1 has m pages, level 2 has n pages, etc.

LRUsim is an interesting application for P2 for many reasons. First, performance is critical: LRUsim runtimes are measured in hours or days. Second, LRUsim's data structures are C++ templates. They are relatively simple structures (e.g., splay trees) that require no sophisticated optimizations. P2 performed well with LEAPS because the retrieval predicates of LEAPS were very complicated and required sophisticated optimizations and manipulations for efficient code to be produced. Not so for LRUsim: the retrieval predicates only specified primary keys. Thus, there was no need for predicate manipulation; there were no obvious opportunities for optimization.

Looking at this problem from another perspective, LRUsim's data structures were implemented as templates (i.e. compositional components). For such a simple application, the use of a transformational generators (P2) seemed to be overkill. Yet, initial results suggested otherwise: our P2 version of LRUsim ran

30% faster than the C++ version (see Table 1). Although LRUsim had undergone extensive tuning by the OOPS group, we discovered a performance bug. Upon correcting the C++ version of LRUsim, benchmarks indicated that the P2 version was still about 6% faster (see right-most columns of Table 3).[5] Thus, even in situations where few or no optimizations are applied, the performance of the software produced by P2 was very good (i.e., comparable to programs hand-coded and tuned by experts).

Trace File	**# of memory references**	**LRUsim_{P2} run-time (sec)**	**$\text{LRUsim}_{C++original}$ run-time (sec)**	**Speedup over** $C++original$	**$\text{LRUsim}_{C++corrected}$ run-time (sec)**	**Speedup over** $C++corrected$
`tex`	200K	100	130	30%	108	8%
`spice`	200K	115	153	33%	124	8%
`ghostscript`	107M	64K	84.2K	31%	66.3K	4%
`expresso`	592M	378K	482K	27%	402k	6%

TABLE 3. LRUsim Execution Times

We learned important lessons from this experiment: synthesized algorithms have a higher probability of performing better (and for the same reasons are more reliable) than hand-coded algorithms. Remember, we are not using formal methods to generate software, nor are we synthesizing customized algorithms (as in [Smi90]), but merely gluing pre-written algorithms together. So our approach to software synthesis is quite simple.

The reason for improved performance is that the authors of generator components focus their attention on coding the most efficient implementation of a set of algorithms for a highly constrained problem. (In the case of P2, these are algorithms for a particular data structure). Consequently, much more effort is focussed on optimizing what would otherwise be called "minute" coding details.

Coding a data structure from scratch that corresponds to a composition of P2 components typically requires far too many details to keep straight and to optimize. It is impractical and too time consuming for programmers to optimize code at the same level of detail. Programmers have finite time and energy, and attempt only a fraction of these optimizations. For this reason, the synthetic algorithms manufactured by P2 will often outperform their hand-written counterparts.

Stated another way, P2 components generate locally optimal code. Composing locally optimal code fragments doesn't guarantee global optimality. So it is always possible to hand-craft code that will out perform that which is produced by a generator. But the issue is "At what cost?". Just as it is possible for humans to outperform optimizing compilers by hand-coding assembly statements, it is generally not considered practical; the quality of compiler output is good enough compared to the productivity advantages gained in programming in higher-level languages. So too it seems for using generators.

Components represent repositories for "best practice" approaches for building software: they can encapsulate coding tricks and proven-effective approaches for implementing their "feature" in an efficient way. Over time, additional effort is put into components so that the code that they produce improves even further, so that it is better (on average) than anything individuals or even groups of programmers can produce. Thus, composing locally optimal components produces very good software.

5. While 6% might not seem significant, notice that the execution times for LRUsim are enormous. The **`expresso`** trace file takes days to complete; a 6% improvement meant the P2 version finished 6 hours faster than the C++ LRUsim [Jim97].

It is interesting to note that there were productivity improvements in building LRUsim with P2. The C++ version of LRUsim is 6600 lines; the P2 version is about 2500 lines. The largest P2 function has 52 lines of code; P2 expands this to 1300 lines. Once again, by raising the level of abstraction, the complexity of an application is substantially reduced. In our exit surveys with the OOPS group, they believe that the ability to compactly write complicated code is P2's major benefit. It enabled revisions and restructuring that they believed would be difficult or impossible to perform by hand.

Our LRUsim experiment also provided us with a valuable data point for choosing between a compositional and transformational implementation of GenVoca refinements. In particular, this experiment showed us that performance should be not a deciding factor in choosing between transformational and compositional implementations. Other factors are more critical; what exactly are these factors is a subject of on-going work.

4.3 ADAGE

ADAGE is a DARPA-sponsored project from 1992-1995. It is a static, compositional generator for avionics software. The following results were reported by Holmquist, Tracz, and Selby [Hol96].

An evaluation of ADAGE was conducted by performing experiments to build navigation software with and without ADAGE. The overall conclusions were (roughly) a factor of 14 improvement in productivity using ADAGE, with an 8:1 reduction in errors.

The improvement in productivity was due to several factors: (1) The configuration process was repeatable: it is a process of selecting components and parameter values. (2) The composition process is repeatable: parameter substitution, component instantiation were automated. (3) Type equations (called decision trees) structured configurations and formalized the process of instantiation. Stated a different way, the process of instantiating and composing components given a graphical specification of a type equation was automated. Consequently, it was done correctly and quickly. This alone led to an improvement in productivity.

At the same time, one can also see how the reduction in errors was accomplished. (1) There was automatic constraint checking: selecting incorrect components and parameter values are detected automatically (just as incorrect configurations and parameter values are detected in P2). (2) There was automatic, not manual, source code modification: this is an example of "intelligent components". (3) Errors were found at specification time, not after the system has been built.

We can add to the observations of the ADAGE team with comments on software reliability. We believe that there was the same density of errors in our P2 component code as any other hand-written application. However, we noticed errors surfaced much faster. The reason is that components can be used in widely varying settings. As a consequence, component code is more thoroughly exercised, thereby revealing errors that might otherwise remain latent. We used a set of regression tests/programs whose correct output was known. Each of these programs could use any one of a number of type equations in which a new component could be inserted. No matter which type equation was chosen, the same output had to be generated. That is, the result of a query is invariant to the way in which elements are stored. The different type equations allowed us to vary the container implementation (and hence the conditions under which the new component was being used), thereby exercising different parts of a component's code.

From our experience, we believe that component technologies will be inherently *more* reliable than traditional software. Using standard testing techniques today, coupled with additional approaches that we have used in evaluating components, we expect that at shipment time, components will be measurably more reliable than typical software.

5 Future Work

Designing components and building generators is difficult. We have made significant progress in understanding how components can be designed and recognized; there are methodologies now that can be followed with good success. However, building these components and their generators remains a challenge. There is little or no tool support, especially for building transformational generators. We estimate that 60% or more of the effort in building a generator and its component libraries focuses on the design and construction of languages for component definitions, component parameterizations, compositions, and automatic checking. We have found that by the time that an adequate infrastructure is in place, and it is time to seriously test the capabilities of a generator, funding has expired. Tools are needed to help build generators to reduce this overhead.

Our current project is *Jakarta*, a tool suite for constructing generators. We are building compiler technologies for creating specification languages for domain-specific applications, and using these same tools for making programming languages extensible.

Our tool suite is based on the Java language. We are now building an extensible version of Java, also called *Jakarta*, that allows us to add new programming constructs (such as new data types that were introduced into C by P2) and to provide language support for expressing programs as data — i.e., abstract syntax trees — which is essential for writing program transformations. We expect Jakarta to be suitable for building both compositional and transformational generators. Compositional components will be Jakarta template libraries or binary libraries. Transformational components will be plug-and-play extensions to the Jakarta language/compiler. Thus, just as we are building customized software systems from components, we will be building customized versions of Jakarta from components.

6 Conclusions

Intelligent components will be a centerpiece for future tools of software development; they automatically perform customizations and optimizations that are specific to the system in which they are used — modifications that today are performed by hand by domain experts. By automating modifications, we have found that intelligent components make "as-is" software reuse practical, while maintaining software quality and reliability without sacrificing application performance.

In this paper, we have reviewed a number of experimental results using generators. Although the actual experiments were rather different, there were common themes in the lessons learned. Generators allow users to program in terms of domain abstractions: this not only eliminates enormous amounts of implementation details but also promotes clean, efficient, and understandable designs of applications. Generators simplify maintenance and application evolution: one can easily make modifications to application code by regenerating the portion that must change. Because of the way components are written, generated code is often more efficient than that produced by hand; this is similar to the situation that we have today where optimizing compilers produce efficient assembly code. Although it is possible to write more efficient assembly code by hand, it generally isn't done because of the benefits offered by programming in higher-level languages. One can understand generators as being next-generation compilers, where the "assembly languages" for generators are conventional languages like C or C++.

Generator technologies are not free. They require programmers and software designers to think about software not in terms of lines of code, or in terms of object-oriented classes, but rather at an architectural level where refinements are the basic building blocks. We believe that the key problems of generators are not technical in nature. To be sure, plenty of technical obstacles remain, but we are confident that they can be overcome. We believe the most difficult challenges are political and organizational: it requires a major shift in organizational thinking to automate the development of software that is presently coded by hand. However, only until software development is automated will major benefits in productivity, quality, reliability, and performance be possible.

Acknowledgements. I thank Yannis Smaragdakis (University of Texas) and Jacques Meekel (Motorola) for their comments on an earlier draft of this paper.

7 References

[Alt96] S. Altschuler and B. Kim, "Generating Signal Processing Software", Microsoft Workshop on Component-Based Software Development, June, 1996.

[Bat88] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise, "Genesis: An Extensible Database Management System", *IEEE Transactions on Software Engineering*, November 1988, 1711-1730.

[Bat92] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 4, October 1992, 355-398.

[Bat93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, "Scalable Software Libraries", *ACM SIGSOFT*, December 1993.

[Bat94a] D. Batory, "The LEAPS Algorithms", Department of Computer Sciences, University of Texas at Austin, Technical Report 94-28.

[Bat94b] D. Batory, J. Thomas, and M. Sirkin, "Reengineering a Complex Application Using a Scalable Data Structure Compiler", *ACM SIGSOFT*, December 1994.

[Bat96] D. Batory and B.J. Geraci. "Composition Validation and Subjectivity in GenVoca Generators". To appear in *IEEE Transactions on Software Engineering*, special issue on Software Reuse. Also, "Validating Component Compositions in Software System Generators", *Fourth International Conference on Software Reuse*, Orlando, Florida, April 1996, 72-83.

[Bat97] D. Batory and J. Thomas, "P2: A Lightweight DBMS Generator". To appear in *Journal of Intelligent Information Systems*.

[Bax92] I. Baxter, "Design Maintenance Systems". *CACM* April 1992, 73-89.

[Big87] T. Biggerstaff and C. Richter, "Reusability Framework, Assessment and Directions", *IEEE Software*, March 1987.

[Big94] T. Biggerstaff, "The Library Scaling Problem and the Limits of Concrete Component Reuse", *Proceedings of the Third International Conference on Reuse*, November 1994.

[Bra93] D. Brant and D. Miranker, "Index Support for Rule Activation", *ACM SIGMOD*, May 1993.

[Cli91] W. Clinger and J. Rees (editors), "The Revised Report on the Algorithmic Language Scheme". *Lisp Pointers IV(3)*, July-September 1991, 1-55.

[Cog93] L. Coglianese and R. Szymanski, "DSSA-ADAGE: An Environment for Architecture-based Avionics Development", *Proceedings of AGARD 1993*.

[Coo88] T. Cooper and Nancy Wogrin, *Rule-based Programming with OPS5*, Morgan-Kaufmann, 1988.

[Gar95] D. Garlan, et al, "Architectural Mismatch or Why It's Hard to Build Systems out of Existing Parts", *ICSE 1995*.

[Gog86] J. Goguen, “Reusing and Interconnecting Software Components”. *Computer*. February 1986.

[Gog96] J. Goguen, “Parameterized Programming and Software Architecture”, *Fourth International Conference on Software Reuse*, Orlando, Florida, April 1996, 2-10.

[Gri94] M.L. Griss and K.D. Wentzel, “Hybrid Domain-Specific Kits for a Flexible Software Factory”, *Proc. ACM SAC’94*, March 1994, 47-52.

[Hei94] J.S. Heideman and G.J. Popek, “File-System Development with Stackable Layers”, *ACM Transactions on Computer Systems*, February 1994.

[Hol96] L. Holmquist, W. Tracz, and R. Selby. “An Evaluation of DSSA-ADAGE Technology”, *Crosstalk*, December 1996, Volume 9, No 12, 25-28.

[Hut91] N. Hutchinson and L. Peterson, “The *x*-kernel: an Architecture for Implementing Network Protocols”, *IEEE Trans. Software Engineering*, January 1991.

[Jim97] G. Jimenez-Perez and D. Batory, “Memory Simulators and Software Generators”. To appear in *1997 Symposium on Software Reuse*.

[Kic91] G. Kiczales, J. des Rivieres, and D.G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

[Kie96] R. Kieburtz, L. McKinney, J. Bell, J. Hook, A.Kotov, J. Lewis, D. Oliva, T. Sheard, I. Smith and L. Walton, “A Software Engineering Experiment in Software Component Generation”. *International Conference on Software Engineering*, 1996.

[Mir90] D. Miranker, D. Brant, B. Lofaso, and D. Gadbois, “On the Performance of Lazy Matching in Production Systems”, *Proc. National Conference on Artificial Intelligence*, 1990.

[Nei89] J. Neighbors, “Draco: A Method for Engineering Reusable Software Components”. In *Software Reusability*, T.J. Biggerstaff and A. Perlis, eds, Addison-Wesley/ACM Press, 1989.

[Per92] D.E. Perry and A.L. Wolf, “Foundations for the Study of Software Architecture”, *ACM SIGSOFT Software Engineering Notes*, October 1992, 40-52.

[Pri91] R. Prieto-Diaz and G. Arango, *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, 1991.

[Sel88] R. Selby, “Empirically Analyzing Software Reuse in a Production Environment”. *Software Reuse — Emerging Technology*, editor Will Tracz, IEEE Computer Society, New York, 1988, 176-189.

[Smi90] D.R. Smith, “KIDS: A Semiautomatic Program Development System”, *IEEE Transactions on Software Engineering*, Sept. 1990, 1024-1043.

[Sim95] C. Simonyi, “The Death of Computer Languages, the Birth of Intentional Programming”. *NATO Science Committee Conference,* 1995.

[Sma97] Y. Smaragdakis and D. Batory, “DiSTiL: a Transformation Library for Data Structures”, in preparation.

[Spr90] G. Springer and D.P. Friedman, *Scheme and the Art of Programming* MIT Press and McGraw Hill, 1990.

[Ude94] J. Udell, “Componentware”, *BYTE*, May 1994.