IEEE Software, September 1994.

Achieving Reuse With Software System Generators[†]

Don Batory, Sankar Dasari, Bart Geraci,
Vivek Singhal, Marty Sirkin, and Jeff Thomas
Department of Computer Sciences
The University of Texas
Austin, Texas 78712

A key problem in software engineering is building complex software systems economically. We believe that domain-specific software system generators is a promising technology for attacking this problem. Generators are realizations of domain models that explain how software systems in a target domain can be assembled from previously written components. Thus, generators require significant problems in software reuse to be solved. In this paper, we review a related set of projects that we have undertaken to understand better the unusual software design techniques that are required and to evaluate the productivity and performance potentials of software system generators.

1 Introduction

Software systems are becoming progressively more complex and expensive to build. Coding systems from scratch, with minimal leverage from one system to the next, clearly is not cost effective and is not a scalable means of construction. Finding more economical ways of building software is a basic goal of software engineering. Software reuse is widely believed to be a key in achieving this goal.

We believe that domain-specific *software system generators* will be indispensable tools of future software development environments. Software system generators are among the foremost technical achievements in software reuse and software architectures: they are realizations of domain models (or reference architectures) that define how software systems of a particular domain can be assembled rapidly by composing components from reuse libraries. The primary goal of generators is to eliminate the mundane aspects of software construction and to permit the expenditure of proportionally more effort on the critical parts of target systems. McIlroy called this goal the "industrialization of software" [McI68].

Our research has focused on domain-specific software system generators. Our early work was Genesis, the first generator for database management systems (DBMSs) [Bat88, Bat92a]. Genesis demonstrated that complex and customized DBMSs could be synthesized in minutes by composing prefabricated and plug-compatible components. The DBMSs that were produced were of university-quality; they were untuned, but substantial in size (i.e., exceeding 60K lines of C code). Genesis was a proof-of-concept; it did not demonstrate that synthetic DBMSs could be efficient. Our current project is Predator; it is a data structure generator that can be understood as a next generation, high performance version of Genesis. A primary goal of this project is to demonstrate that high quality generators that rely on reusable component libraries offer substantial productivity and performance gains over handwritten and hand-optimized code.

Our experiences with domain-specific software generators are not unique. Similar experiences have been noted and virtually identical software organizations have been used in independently-conceived generators for the domains of network protocols [Oma92], data manipulation languages [Vil94], distributed file sys-

[†]This research was supported in part by grants from the University of Texas Applied Research Laboratories, Schlumberger, and Digital Equipment Corporation.

tems [Hei93], host-at-sea buoy systems [Wei90], and real-time avionics software [Bat93b]. Thus it seems worthwhile to factor out the common, domain-independent ideas that underlie different software system generators, and to build tools and develop design techniques that support these particular methods of software organization and construction. By doing so, we believe that other researchers in software engineering can benefit from these collective experiences without having to delve into the obscure details and vagaries of the particular domains from which they came.

In this paper, we review some of our current projects which we hope will enable other researchers to understand the principles of software system generators and reusable software that have implicitly guided our research efforts over the last ten years. We also present experimental evidence that shows software system generators can significantly enhance productivity and produce software whose performance is comparable to that written by hand. We conclude by exploring the challenges ahead and suggest future areas of research that seem promising.

2 The Predator projects

Research in software reuse and software system generators is difficult and challenging for several reasons. First, creating a software generator for any target domain requires a thorough understanding of that domain and how software in that domain has been built. Acquiring domain expertise in order to formulate a realistic domain model requires a considerable investment on the part of reuse researchers [Cur88].

Second, it is not sufficient to propose a domain model; the model must be validated through extensive prototyping. We estimate that building a software system generator requires 50% to 100% more effort than building a single system. This overhead is easy to understand: assembling a single system from components is not impressive; one needs to build components for at least two complete systems to demonstrate the payoff (i.e., the mix-and-match capabilities) of a generator.

Third, domain-specific results (i.e., generators) are usually of interest only to a small community of researchers. For a domain-specific result to be accessible and appreciated by the general reuse community, the design principles used must be identified and represented in a domain-independent way. Defining domain-independent abstractions that are central to reuse is often a formidable intellectual undertaking.

From our experience, the only way that people can truly understand software reuse and software system generators is through hands-on experience. Data structures is among the few domains that all software engineers consider themselves experts; it is a microcosm of the problems that exist in domains of recognizably large and complex software systems. Therefore, we have embarked on the development of two prototype generators for data structures, P1 and P2, to demonstrate to a wide audience the capabilities and potential for domain-specific generators. (P1 was intended to be an experimental prototype; a more modular version, P2, is intended for distribution). Productivity and performance results on P1 and P2 are presented in Section 5.

As mentioned earlier, explaining how domain-specific results follow from an application of domain-independent concepts is essential to software system generator promulgation. The next section (Section 3) describes domain-independent language extensions to C++ that expose the concepts that are critical to our approach to domain modeling, software reuse, and software system generation; these extensions also distinguish the software organizations of generators from traditional object-oriented design and language concepts. We will use a running example in the next two sections, first to explain some important syntactic concepts and then to define their semantic interpretations for the domain of data structures.

3 GenVoca and P++

As mentioned earlier, our experience with software system generators is not unique. Our analysis of several existing generators (e.g., Genesis - DBMSs, Avoca - network protocols, Ficus - distributed file systems, Brale - host-at-sea buoy systems) revealed the following similarities:

- Subsystems are the building blocks of generated systems.

  Effective software system synthesis requires that systems be constructed from combinations of subsystems (or components), not just functions or classes. It is too unwieldy to construct a large software system by selecting and assembling hundreds or thousands of functions and classes from a reuse library. Instead, larger units of software encapsulation are needed [Joh88].

- Components import and export standardized interfaces.

  The key to software system synthesis is composition. Composition is much easier when component interfaces correspond to fundamental abstractions of the target domain and these interfaces have been standardized. Standardization encourages functionally similar components to be plug-compatible and interchangeable.

- Component composition and customization is achieved through parameterization.

  Parameterization is an easy-to-understand model for combining and customizing components. Simple forms of parameterization, i.e., constant and type parameters, are necessary but not sufficient for software system generators. Components must also be able to import other components as parameters [Gog86, Tra93].

These lessons on software design have been captured in GenVoca [Bat92b]. It is a domain-independent model for defining families of hierarchical systems as compositions of reusable components. From our experience, models of software design are best understood when programming languages provide direct support for model constructs. Thus, we are developing the P++ language, an extended version of C++, to clarify GenVoca concepts and to distinguish them from traditional object-oriented concepts. The central features of P++ are outlined in the following paragraphs.

In GenVoca, the basic unit of software system construction is the *component* or *subsystem*. A component is a suite of interrelated variables, functions, and classes that work together as a unit to implement a particular feature of a software system for a given problem domain. A *realm* is a library of plug-compatible components. A realm is defined by a standardized interface that consists of functions and classes. All components of a realm inherit this interface and may specialize it by adding data and function members to existing classes, and by adding new variables, functions, and classes.

P++ has linguistic constructs for defining realms and components. Figure 1a shows a definition for realm **`DS`** which consists of classes **`container`** and **`cursor`**. (For the moment, we ask readers to forego assigning meanings to the examples in this section, and focus on the syntactic/abstract descriptive capabilities of P++; we will explain the semantics of this realm and its components in Section 4). Figure 1b shows the **`transient`** component, an instance of the **`DS`** realm. Note that **`transient`** specializes the interface of **`DS`** in three ways: (1) classes **`container`** and **`cursor`** have additional data and/or function members; (2) a new class, **`element`**, has been added, and (3) new variables, **`top_of_heap`** and **`free_bytes_left`**, are present.

Parameterization is a key feature of P++. Realms may be parameterized by types and constants; all realm parameters are inherited by the realm's components. Notice that realm **`DS`** in Figure 1a is parameterized by class **`e`**. The **`transient`** component inherits parameter **`e`** and uses it to define class **`element`**.

```
realm DS <class e>
{
  class container
  { container (); };

  class cursor
  {
    cursor (container *c);
    void move_to_start ();
    void advance ();
    void insert (e& obj);
    void remove ();
    bool end_of_container ();
    ...
  };
};
```

```
component transient : realm DS<class e>
{
  class container
  { bool opened (); };

  class cursor
  { element *obj; };

  class element : class e
  { ... };

  char *top_of_heap;
  int  free_bytes_left;
};
```

Figure 1a and 1b: A sample realm and component.

```
component bintree<member m,DS<element> x>
     : realm DS<class e>
{
  class container
  { element *root_element; ... };

  class element : class e
  {
    element *left_child;
    element *right_child;
    ...
  };
};
```

```
component hash<const int h, member m,
     DS<element> x> : realm DS<class e>
{
  class container
  { element *bucket[h]; ... };

  class element : class e
  { element *next_on_hash_chain; ... };

  int hash ()
  { ... };
};
```

Figure 2a and 2b: Parameterized components.

Individual components may be additionally parameterized by realms, classes, data member names, and constants. Figure 2a shows the **DS** component **bintree**, which is parameterized by data member **m** and **DS** component **x**. Figure 2b shows component **hash**, which has three parameters: constant integer **h**, data member **m**, and **DS** component **x**.

A software system specification is expressed in P++ by composing components through parameter instantiation. Suppose **emp** denotes a class of employees which has members **age** and **name**. The following compositions are examples of P++ software system specifications:

```
typedef hash<emp> <1000, age, transient> k0;
typedef bintree<emp> <age, hash <500, name, transient>> k1;
```

Type **k0** defines an implementation of the **DS** interface where the **hash** component is layered on top of **transient**. Additional parameters to **hash** are **1000** and data member **age**. **<emp>** is separately listed because it is a parameter for realm **DS**. Type **k1** defines an implementation which stacks the **bintree**, **hash**, and **transient** components (in this order). Data member **age** is a parameter to **bintree**, while **500**, the member **name**, and the component **transient** are parameters of **hash**.

The semantics of component parameters and composition is domain dependent. In the following section, we present a model of the data structure domain which makes these compositions meaningful.

4 The data structures domain and the Predator generators

As mentioned in Section 2, data structures is among the few mature domains that are well-understood by all software engineers. General-purpose tools for data structure generation are virtually nonexistent; most data structures today are still coded by hand. Because the data structure domain typifies the problems that are encountered in domains of recognizably large software systems, and because opportunities for using generators are abundant, data structures is an ideal domain to study. Although our prototyping efforts (P1 and P2) have focussed on extensions to ANSI C, we will use P++ in this section to express the basic concepts of our domain model.

4.1 A domain model for data structures

Most common data structures – binary trees, lists, arrays – are implementations of a container abstraction. A *container* is a sequence of *elements*, where all elements of a container are instances of a single class. Elements of a container are enclosed by the container and can be referenced and modified only through runtime objects called *cursors*. Readers may recognize cursors and containers as well-established concepts in databases [Kor91]; our earlier work on Genesis and contemporary work in object-oriented databases strongly influenced our choice of these abstractions [ACM91]. Figure 3a illustrates the basic ideas; a container with eight elements is shown, with a cursor that references one of these objects.

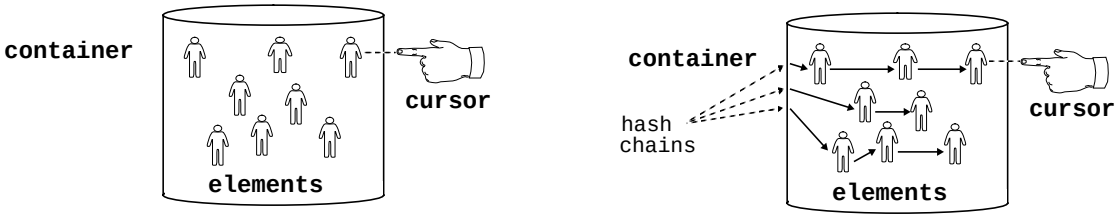

Figure 3a and 3b: DS abstractions and component mappings.

DS is the realm of components that implement the container-cursor abstractions. Programs that reference the **DS** interface see only **container** and **cursor** objects, yet **DS** components must explicitly define **element** objects in order to implement container-cursor abstractions. This latter point can be seen in the **transient**, **bintree**, and **hash** components of Figure 1 and Figure 2.

Every **DS** component behaves like a transformation that implements the objects and operations of the **DS** interface. As an example, recall the **hash<h,m,x>** component of Figure 2b. **hash** links **elements** with the same hash value onto a common bucket chain. The **m** parameter represents the name of a data member of type **e** that is to be hashed, and **h** is the total number of bucket chains. Each object of type **e** is specialized by the addition of a data member (**next_on_hash_chain**) to implement a bucket list. Figure 3b illustrates these ideas.

Note that **hash** is a symmetric transformation, i.e., it maps the container-cursor abstraction to concrete implementations of the container-cursor-element classes, which in turn can be further transformed by other **DS** components. Symmetry is expressed by **hash**'s realm parameter **x**, which indicates how hash-specialized elements (denoted by **DS<element>** in Figure 2b) are to be mapped by some other **DS** component **x**.

In general, **DS** components encapsulate basic data structure algorithms and generic data structure features (e.g., sequential and random storage, storage in persistent or transient memory, etc.). Using P++ syntax, a library of **DS** members can be easily defined:

```
typedef DS<element> ds;                                  // ds is an abbreviation

component odlist<ds d, member m>: realm DS<class e>;  // doubly linked list sorted
                                                         // on member m
component dlist<ds d>         : realm DS<class e>;    // unsorted doubly linked list
component array<ds d, int s>  : realm DS<class e>;    // store elements in an array
                                                         // of size s
component malloc<ds d>        : realm DS<class e>;    // store elements on a heap
component persistent<char *f> : realm DS<class e>;    // store elements in a file f
```

As mentioned earlier, compositions of components define software systems, which in our domain corresponds to complex data structures. Recall the compositions **k0** and **k1** from last section, which we repeat below:

```
typedef hash<emp> <1000, age, transient> k0;
typedef bintree<emp> <age, hash <500, name, transient>> k1;
```

Type **k0** defines a data structure where **emp** elements are hashed on member **age** and stored in transient memory. Type **k1** defines a data structure where **emp** elements are first linked onto a binary tree using data member **age** as the key. Next, the binary-tree-specialized elements are hashed on member **name** and stored in transient memory. As these examples suggest, **DS** components can be composed and reused in many ways to produce a great number of complex data structures. Examples are given in Section 5.

In addition to the container-cursor-element abstractions, Predator also provides link abstractions. A *link* is a relationship between elements of (possibly distinct) containers. Well-known implementations of links are relational join algorithms and pointer-based methods (e.g., ring lists) used in object-oriented databases [Kor91, ACM91]. We will not elaborate more on links or the realm of components that implement them. The basic idea that we wish to convey is that our domain model relies on database-like interface to data structures. We believe these abstractions are general enough to realize a software generator for this domain and to make data structure programming much easier.

4.2 The Predator generators

The Predator prototypes P1 and P2 extend the ANSI C language. Both are implemented as translators which transform Predator programs into ANSI C programs; Predator declarations and functions are replaced with their corresponding ANSI C implementations.

Generating efficient code is a primary goal in the design of these prototypes. Because exported functions of components tend to be small, eliminating function call overhead is a key concern. Both P1 and P2 rely on macro expansion and partial evaluation to optimize generated code and to remove the explicit boundaries between components. Presently, P2 is over 30K lines of C code with a library of 40 components. A sizable fraction of P2 implements features that will be eventually provided by P++. P++ will be the platform for our future development efforts. Further details on the architecture and optimizations of these prototypes is given in [Bat92c, Sin93, Sir93].

5 Results

For any generator to be practical, it must satisfy two requirements: (1) use of the generator should yield significant productivity gains, and (2) the performance of generated code must be comparable to that of handwritten code. Some of our findings on these topics are presented in the following sections.

5.1 Generators versus class libraries

Class (or template) libraries are a popular means of boosting programmer productivity and reducing the time and cost of software development. The Booch Components [Boo87], libg++ [Lea88], NIHCL [Gor90], and COOL [Fon90] are examples of such libraries. A large fraction (i.e., over 80%) of these libraries is devoted to components for generic data structures (e.g., lists, trees, queues, etc.). These components offered a prime opportunity for evaluating Predator.

Our first experiment was to benchmark three spell-checker programs: one implemented using Predator, a second using Booch Components, and a third using libg++. Overall, we were pleasantly surprised that the performance of Predator-generated code was marginally superior to (and in some cases, significantly better than) code written for class libraries. The big win, however, was in programmer productivity. By changing **`DS`** component compositions, we were able to alter the data structure implementations of the Predator spell-checker immediately. In contrast, substantial effort was needed to recode spell-checkers to use different Booch or libg++ components. Details of these experiments are presented in [Bat93a].

We learned three valuable lessons from these experiments. First, it is possible to generate code for simple data structures that is comparable in performance to hand-crafted and hand-optimized code. Second, current template libraries are not designed to maximize programmer productivity. Every template library consists of families of related components. While members of a family may share the same interface, members of different families will have incompatible interfaces. That is, it is very common for several families (e.g., binary trees, lists, arrays) to each implement the same abstraction but be given different interfaces. This limits component interchangeability and makes libraries harder to use. A programmer must invest substantial effort to learn and program different interfaces. Predator, however, offers a single standard interface for all elementary data structures which leads to higher productivity.

Third and most important, most template libraries are unscalable. To our knowledge, most data structure libraries reflect feature combinatorics; that is, every component in a library represents a unique combination of features. For example, there are $3 \times 3 \times 2 = 18$ varieties of deques (double-ended queues) in the Booch library. They are derived from the cross-product of 3 concurrency control features (sequential, guarded, synchronized), 3 memory allocation features (bounded, unbounded, dynamic), and 2 ordering features (ordered, unordered). Adding new features, in general, may cause libraries to double in size. For example, all data structures in the Booch library reside in transient memory. If a persistent memory option is added, the library doubles (i.e., 18 transient deques, 18 persistent deques). The problem is actually worse: data structures used in operating systems, compilers, and database systems are far more sophisticated than those available in contemporary template libraries. These applications would require highly specialized features to be frequently added to libraries. Handwritten template libraries are clearly not the solution. Predator offers an attractive alternative: its components implement primitive features and Predator is the tool for composing components. By making feature combinatorics explicit, it is possible that a practical and scalable tool for data structure generation can be realized. This prospect led to our next series of experiments.

5.2 A re-engineering of OPS5/C LEAPS

OPS5/C LEAPS is a state-of-the-art production system compiler that translates complex OPS5 rule sets into C programs [Mir91, Bra92-93]. The run-time efficiency of LEAPS-generated programs has been documented to be several orders of magnitude faster than that of OPS5 interpreters. In addition to the obvious increase in performance by compilation, most of the gains are due to the use of special algorithms for rule processing that avoid materialization of intermediate matches and that rely on sophisticated data structures. The LEAPS data structures implement a main-memory database of assertions that are probed by highly-optimized temporal queries. This database uses predicate and attribute indices, unusual structures for identifying negated working memory elements, and a special implementation of nested loops to process (relational) joins of containers.

LEAPS is a complex, performance-driven application. It was an ideal next target for Predator for three reasons. First, we knew that each of the data structure features needed by LEAPS (e.g., temporal predicates, attribute indices) were certainly not part of any template library; LEAPS presented an acid test for data structure scalability. Second, the primary goal of the LEAPS compiler was performance; head-to-head comparison of run-times would reveal the quality of generated code for non-trivial data structures. Third, it was well-known that the monolithic design of OPS5/C made experimentation with internal data structures very difficult; such experimentation – although difficult – has been vital to the continued improvement of the compiler. Thus, the quality and extensibility of a Predator-generated system would be tested.

LEAPS took approximately three person-years to build. Re-engineering LEAPS took approximately five-person months: two months to understand the LEAPS algorithms and data structures and another three months to build the additional components needed for the LEAPS data structures. Initially, OPS5 rule sets were translated by hand into P1 programs; we have since written a translator to automatically generate P1 programs given an OPS5 rule-set. (This translator took six-person months to build.)

We used a variety of rule sets to benchmark LEAPS and our re-engineered version, called RL. Starting with simple rule sets and small numbers of stored objects, and then progressing to more complex rule sets, we compared RL run-times with those of LEAPS. We were surprised to find that with virtually no exceptions, the running times of RL-generated programs performed at least 10% better than their LEAPS counterparts. Figure 4 displays the differences in performance for two simple rule sets: **`triples`** and **`bigjoin`**. **`triples`** is a rule set for generating all triples **`<x,y,z>`** of integers less than **`n`**, where **`x`** < **`y`** < **`z`** < **`n`**. **`bigjoin`** is a rule set that performs a database join on two containers of **`n`** objects. Both the RL and LEAPS implementations used very complicated data structures to solve the **`triples`** and **`bigjoin`** rule sets. Performance results on more complex rule sets are presented in [Sir94].

The productivity gains using Predator became clear when we altered RL to store its data in persistent memory. This took two days for us to write and debug a **`persistent DS`** component; to swap it with the **`transient DS`** component took minutes. We compared persistent-RL to DATEX, the persistent-memory version of LEAPS [Bra93]. DATEX was a reimplementation of LEAPS using the Genesis file manager [Bat88] and took many months to build. Interviewing members of the LEAPS team suggests that Predator offers a factor of 3 in productivity that leads to a more extensible product. Moreover, since the Genesis file manager was never tuned, we had expected a 50-fold performance improvement; we measured at least a factor of 60. Details of these experiments are presented in [Sir94].

We believe that the productivity and performance advantages of using Predator stem from three sources. First, decomposing complex monolithic data structures into primitive, independent, encapsulated components significantly simplifies implementation and makes possible performance optimizations that are difficult, if not impossible, to accomplish by hand. Second, the high-level programming abstractions offered by Predator radically simplify software system design and permits programmers to concentrate on more criti-

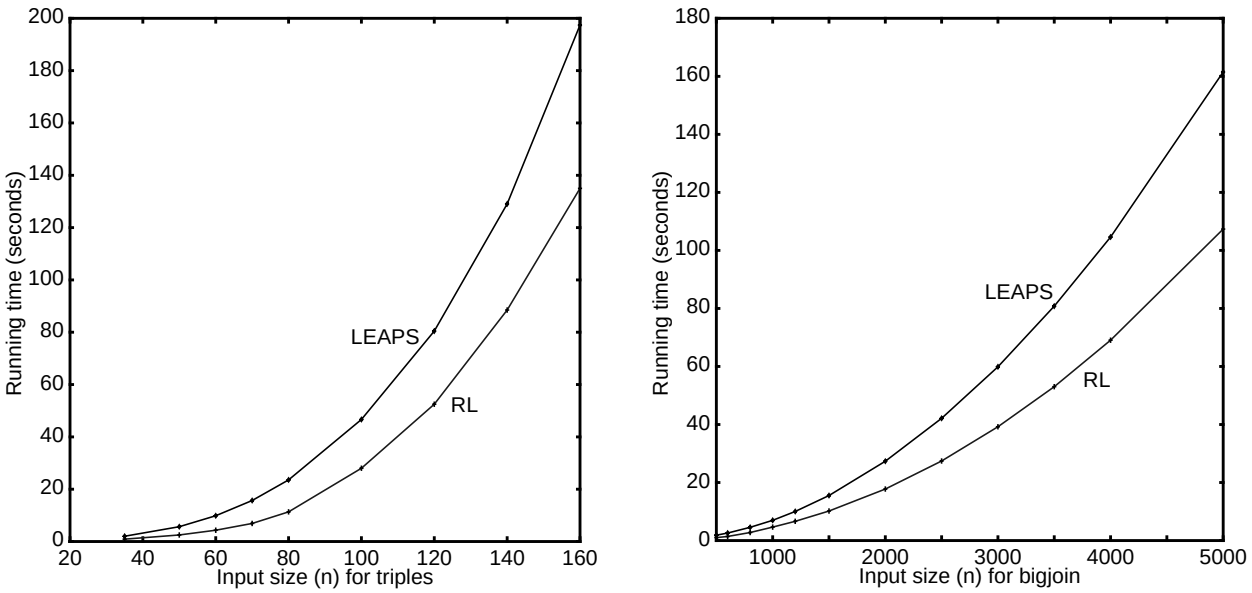

Figure 4: Performance of two rule sets.

cal aspects of software construction. Third, alteration and experimentation with different implementations is facilitated by component compositions.

6 Related work

We noted in earlier sections that the basic ideas of GenVoca can be recognized in the work of many other researchers. Component definition constructs are provided by languages like Ada 9X [Bar93] and Modula 3 [Car89]. Realms and symmetric components can be seen in Volpano's STS [Vol85]. Goguen provides a model for parameterized programming in the languages LIL [Gog86] and FOOPS [Gog93]; the functional language ML [Mil90] takes another approach to parameterization.

Software system generators are related to application generators. Many application generators provide a domain-specific programming language, and include either an interpreter or compiler for that language [Kru92]. Well-known examples of application generators include Lex, Yacc, VisiCalc, Mathematica, and so-called fourth generation languages (4GL's). The reuse advantages of application generators are well-known, affording gains in productivity over general-purpose programming languages for constructing families of related programs. The disadvantages of application generators are (1) limited availability, (2) lack of appropriate functionality, or (3) poor performance [Kru92].

Software system generators provide the advantages of application generators while addressing their disadvantages. In our case, we have noted: (1) GenVoca describes a model and P++ provides a tool for formalizing and simplifying the construction of generators, thus facilitating the development of new generators in other domains. (2) Encapsulating functionality into separate components makes it easier to add new features – the new code can be written without having to understand and modify the existing code. (3) If each component separately implements a different feature, it is easy to select an appropriate set of components to meet specific functional and performance requirements (versus monolithic systems which trade generality for performance).

Draco was one of the first tools for software system generators based on transformation systems. Unlike GenVoca, system designs were expressed in an application-specific language, and were mapped to expressions in other domain-specific languages during the process of code generation [Nei84, Nei89]. Therefore, Draco embodied a different model of application generators than GenVoca does.

7 Conclusions and future work

We are convinced that software reuse is inherently an experimental discipline; the techniques to achieve reuse are best revealed, understood, and learned through experience. Software reuse is often mistaken to be an end-product; rather, it is a by-product in achieving other goals. We have investigated reuse in our quest to define design techniques, languages, and prototypes of domain-specific software system generators.

We chose the domain of data structures as the basis for our research. It is a mature domain that is universally understood; there are numerous applications with handwritten data structures that could be re-engineered and regenerated using a data structure generator. Our preliminary results have been encouraging. We have obtained significant productivity gains without sacrificing run-time performance using our generators. More significantly, we also have evidence that our generated data structure code is more extensible (and hence more maintainable) than handwritten code. However, we have no illusions that our experimental results to date are conclusive; many more experiments (particularly by others using P2) will be needed before a solid case (for or against generators) can be made and the limits of data structure generators are understood.

As mentioned earlier, creating a software system generator for data structures is not the primary emphasis of our work. We are interested in general design techniques to create generators for other domains, where the P1 and P2 prototypes are simply instances of these ideas. The P++ project extends C++ with linguistic features for expressing large-scale components: that is, P++ supports component abstraction, encapsulation, parameterization, composition, and inheritance. These extensions reinforce our belief that design techniques needed for large-scale software reuse and software system generators are inadequately covered by conventional (object-oriented) languages and software design techniques. Here again, we believe that P++ is certainly not the final point for language support for reuse; it is simply a first step. Experience using P++ in software generators is the next order of business.

Acknowledgments. The authors would like to thank Dan Miranker and Kelsey Shepherd for their helpful comments on earlier drafts of this paper.

8 References

[ACM91] ACM. Next generation database systems. *Communications of the ACM*, 34(10), October 1991.

[Bar93] John Barnes. Introducing Ada 9X. Technical Report, Intermetrics Inc., February 1993.

[Bat88] Don Batory. Concepts for a DBMS synthesizer. In *Proceedings of ACM Principles of Database Systems Conference*, 1988. Also in Rubin Prieto-Dmaz and Guillermo Arango, editors, *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, 1991.

[Bat92a] Don Batory and Jim Barnett, DaTE: the Genesis DBMS software layout editor. In Pericles Loucopoulos and Roberto Zicari, editors, *Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development*. Wiley, 1992.

[Bat92b] D. Batory and S.W. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355-398, October 1992.

[Bat92c] Don Batory, Vivek Singhal, and Marty Sirkin. Implementing a domain model for data structures. *International Journal of Software Engineering and Knowledge Engineering*, 2(3):375-402, September 1992.

[Bat93a] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable software libraries. In *Proceedings of the ACM SIGSOFT '93: Symposium on the Foundations of Software Engineering*, December 1993.

[Bat93b] Don Batory, Lou Coglianese, Mark Goodwin, and Steve Shafer. Creating reference architectures: an example from avionics. Department of Computer Sciences, University of Texas at Austin, September 1993.

[Boo87] Grady Booch. *Software Components with Ada*, Benjamin/Cummings, 1987.

[Bra92] D. A. Brant, T. Grose, B. Lofaso, and D. P. Miranker. Effects of database size on rule system performance: five case studies. *Proceedings of the International Conference on Very Large Data Bases*, 1992.

[Bra93] David Brant and Dan Miranker. Index support for rule activation. In *Proceedings of 1993 ACM SIGMOD*, May 1993.

[Car89] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical Report 52, Systems Research Center, Digital Equipment Corporation, November 1989.

[Cur88] Bill Curtis, Herb Krasner, and Neil Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11), November 1988.

[Fon90] M. Fontana, L. Oren, and M. Neath. COOL: C++ object-oriented library. Texas Instruments, 1990.

[Gog86] Joseph Goguen. Reusing and interconnecting software components. IEEE Computer, 19(2):16-28, February 1986. Also in Rubin Prieto-Dmaz and Guillermo Arango, editors, Domain Analysis and Software Systems Modeling. IEEE Computer Society Press, 1991.

[Gog93] Joseph Goguen and Adolfo Socorro. Module composition and system design for the object paradigm. Technical Report, Programming Research Group, Oxford University Computing Laboratory, August 1993.

[Gor90] K. Gorlen, S. Orlow, and P. Plexico. Data Abstraction and Object-Oriented Programming in C++, John Wiley, New York, 1990.

[Hei93] John Heidemann and Gerald Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, to appear. Also in Technical Report CSD-930019, Department of Computer Science, University of California, Los Angeles, July 1993.

[Joh88] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June/July 1988.

[Kor91] Hank Korth and Avi Silberschatz. *Database System Concepts*, McGraw-Hill, 1991.

[Kru92] Charles Krueger. Software reuse. *ACM Computing Surveys*, June 1992.

[Lea88] Doug Lea. libg++, the GNU C++ library. In *Proceedings of the USENIX C++ Conference*, 1988.

[McI68] M. D. McIlroy. Mass produced software components. In Peter Naur and Brian Randell, editors, *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee*, NATO Scientific Affairs Division, October 1968.

[Mil90] Robin Milner, Mads Tofte, and Robert Harper. The Definition of Standard ML, MIT Press, 1990.

[Mir91] D. P. Miranker, D. Brant, B.J. Lofaso, and D. Gadbois. On the performance of lazy matching in production system. In *Proceedings of the 1990 National Conference on Artificial Intelligence*, July 1990.

[Nei84] James M. Neighbors. The Draco approach to constructing software from reusable components. In *IEEE Transactions on Software Engineering*, SE-10(5), September 1984. Also in Peter Freeman, editor. *Software Reusability.* IEEE Computer Science Press, 1987.

[Nei89] James M. Neighbors. Draco: a method for engineering reusable software systems. In Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability, Volume I: Concepts and Models*, ACM Press, 1989.

[OMa92] Sean O'Malley and Larry Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110-143, May 1992.

[Sin93] Vivek Singhal and Don Batory. P++: a language for software system generators. Technical Report TR-93-16, Department of Computer Sciences, University of Texas at Austin, November 1993.

[Sir93] Marty Sirkin, Don Batory, and Vivek Singhal. Software components in a data structure precompiler. In *Proceedings of the 15th International Conference on Software Engineering*, May 1993.

[Sir94] Marty Sirkin. *A Software System Generator for Data Structures*, Ph.D. Thesis. Department of Computer Sciences and Engineering, University of Washington, January 1994.

[Tra93] Will Tracz. LILEANNA: a parameterized programming language. In *Proceedings of the 2nd International Workshop on Software Reuse*, March 1993.

[Vol85] D. Volpano and R. Kieburtz. Software templates, In *Proceedings of the 8th International Conference on Software Engineering*, 1985.

[Vil94] E. Villarreal. *Automated Compiler Generation for Extensible Data Languages*, Ph.D. Thesis. Department of Computer Sciences, University of Texas at Austin, 1994.

[Wei90] David M. Weiss. Synthesis Operational Scenarios. Technical Report 90038-N, Version 1.00.01, Software Productivity Consortium, Herndon, Virginia, August 1990.