Design Wizards and Visual Programming Environments for GenVoca Generators

Don Batory, *Member*, *IEEE Computer Society*, Gang Chen, Eric Robertson, and Tao Wang

Abstract—Domain-specific generators will increasingly rely on graphical languages for declarative specifications of target applications. Such languages will provide front-ends to generators and related tools to produce customized code on demand. Critical to the success of this approach will be domain-specific design wizards, tools that guide users in their selection of components for constructing particular applications. In this paper, we present the P3 ContainerStore graphical language, its generator, and design wizard.

Index Terms—Self-adaptive software, architectural optimizations, generators, components, refinements, applications product-lines.

1 Introduction

Domain-specific languages (DSLs) will become progressively more important as a medium for specifying customized applications [5], [22], [15], [34]. Generators are tools—compilers, really—that convert DSL application specifications into optimized source code. Visual programming languages, such as Visual Basic and VisualAge, will simplify the use of DSLs and promote their promulgation. But, more importantly, visual programming (or, more accurately, visual specification) languages will offer a convenient way to integrate a suite of analysis tools that will substantially enhance the capabilities and effectiveness of generators.

We are exploring the use of a visual specification and analysis environment for a Java-based generator called P3. P3 is a GenVoca (i.e., component-based) generator for container data structures that is a successor to P2 [5], [6]. P3 is a modular extension of the Java language that allows container data structures to be specified declaratively. That is, P3 adds data-structure-specific statements to Java so that users can compactly specify the implementation of a target data structure as a composition of reusable P3 components. The P3 generator, which is actually a Java preprocessor, translates P3 programs directly into pure Java programs. Among the features that make P3 attractive is that it is equivalent to a gargantuan library of container data structures whose efficiency is comparable to (or better than) hand-coded libraries that are now available.

To promote and simplify the use of P3, we have developed the ContainerStore applet as a visual programming language for writing P3 programs. Clients fill in forms and edit diagrams from which the ContainerStore can infer P3 data structure specifications. While the applet itself is not a major innovation, it is interesting because it integrates a suite of tools and services that makes P3 programming more effective—*tools and services that P3 alone cannot provide*. The ContainerStore offers facilities for *explaining* compositions of components so that clients can verify that the data structure that they have defined is the one that they want. If there are errors in a component composition (or in any other phase of specification), they are caught immediately and explanations of how to repair the errors are provided. By far, the most innovative aspect of the ContainerStore tool suite is a prototype technology for automatically critiquing and optimizing container implementations (i.e., P3 component compositions) for a particular workload. Given a set of components and rules that express knowledge of what combinations of components are best suited for solving particular problems, a tool called a *design wizard* applies these rules automatically to critique and optimize a P3 specification. If the design wizard discovers a composition of components that is likely to perform better than that specified by a user, this composition is reported and reasons are given to explain why the alternative composition is an improvement. In this way, design wizards offer expert guidance so that design blunders can be avoided.

In this paper, we present the P3 ContainerStore applet, its generator, and its design wizard.

2 The P3 ContainerStore Applet

The *ContainerStore* is a visual domain-specific language for specifying container data structures. Specifications are disributed across five tabs (i.e., presentation windows/panels) and are completed in sequence (see Fig. 1). The first tab allows a user to specify the class of elements that are to be stored. In particular, the name of the element class, and the name of each attribute, its type, and cardinality are entered.[1] As a running example, suppose elements of class **emp** are to be stored, where **emp** objects have **name** and **age** attributes.

• *The authors are with the Department of Computer Science, University of Texas at Austin, Austin, TX 78712. E-mail: batory@cs.utexas.edu.*

Manuscript received 26 Oct. 1998; revised 29 Apr. 1999; accepted 13 May 1999.
Recommended for acceptance by D.E. Perry.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 110649.

1. The *cardinality* of an attribute is the expected number of distinct values that the attribute will be assigned.

0098-5589/00/$10.00 © 2000 IEEE

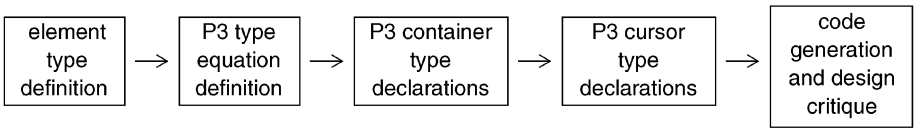

Fig. 1. Sequence of panel specifications in ContainerStore.

The second tab—called the *Type Equation Tab* (Fig. 2)—presents a visual interface for defining customized container implementations as a linear composition of P3 components (also known as a *P3 type equation* or *P3 component stack*). Fig. 2 shows two stacks: the "**Equation**" stack has components **rbtree** (red-black tree) on top of **hash** (hash table) on top of **malloc** (transient heap); the "**Equation2**" stack has **dlist** (doubly linked list) on top of **hashcmp** (hash comparison) and **malloc**. Stacks can be edited (e.g., components can be replaced and deleted) and annotations can be added. An *annotation* is a configuration parameter that is specific to a component. In Fig. 2, annotations to the **hash** data structure component are specified by clicking **hash** and entering the name of the key to hash (**age**) and the number of buckets (**100**) in the *Annotations* fields.

Once a type equation (component stack) has been constructed, the *Explain/View Type Equation* button is pressed. If the equation is valid, an explanation of its meaning is shown in the *Explain Window*. In Fig. 2, the meaning of the "**Equation**" stack is:

> A container of elements of type emp where all elements are stored in ascending name order on a red-black tree and all elements are hashed on age and stored in 100 buckets that are insertion-ordered doubly linked lists in transient memory.

As a general rule, people who are unfamiliar with P3 type equations are unfamiliar with their interpretation. For these users, this facility for explaining equations is invaluable. Explanations are generated using the same techniques that P3 uses to generate Java code; that is, instead of composing code fragments, the explanation is composed from English phrases. In the case that an equation is incorrect—i.e., constraints for the correct usage of a component have been violated (see [7])—the Explain Window lists the errors and suggests reparations. For example, "**Equation2**" is incorrect:

Design Rule Error: move hashcmp above dlist;
Design Rule Error: no retrieval layer beneath hashcmp;

The first error message says that ordering of the **hashcmp** and **dlist** layers is incorrect; a correct ordering would reverse their positions in the equation (the actual reasons are low-level and are not given—only that a correct composition requires **hashcmp** to be above **dlist**). Applying this modification (it turns out) satisfies the objections of the second error message, thus yielding a correct equation. In this way, the ContainerStore gives invaluable guidance to software designers: It helps them repair incorrect compositions and it helps them verify that the specified data structure is indeed the one that they want.

The third tab is where the names of the container classes and their implementing type equations are specified. Suppose container class named **ec** implemented by **Equation** is defined. The fourth tab specifies cursor classes (Fig. 3). A *cursor* is a run-time object that is used to reference, update, and delete elements in a container. In Fig. 3, the cursor class **few** is defined. Its constructor has a single parameter **x** of type **ec**—meaning that every **few** instance will be bound to an instance of container class **ec**. The selection predicate of a cursor class is specified incrementally using the *Predicate Builder*, which allows clauses of the form (attribute relation value) to be declared

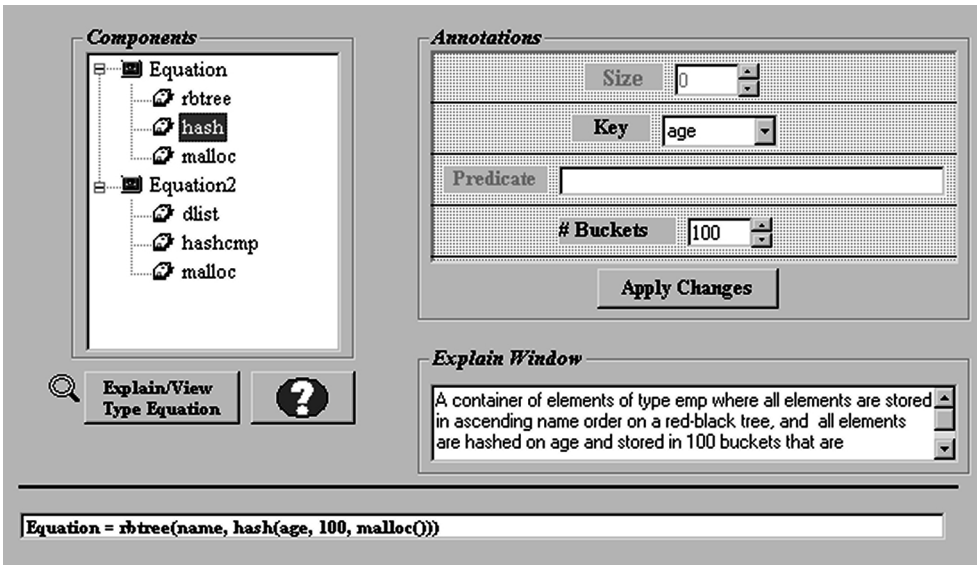

Fig. 2. The Type Equation Tab.

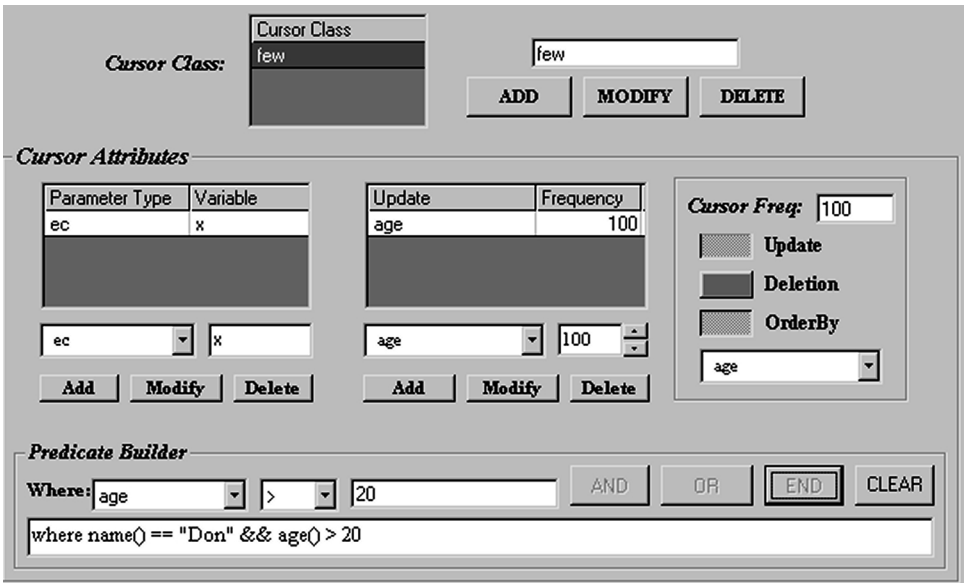

Fig. 3. The Cursor Tab.

separately. The predicate for **few** selects **ec** container elements where attribute **name** is "**Don**" and **age** is greater than **20**. In addition, elements will be retrieved in **age** order (as specified by the *OrderBy* field) and the **age** attribute of selected elements will be updated. The frequency that **few** retrievals are performed is estimated by the user to be **100** times per time period.

Finally, the fifth tab (the Generate tab) presents a scrollable window and four buttons. One button generates the P3 specification of a ContainerStore declaration; a second button invokes the P3 generator to convert this specification into Java source. The generated source is displayed in the scrollable window. (Section 3 illustrates both specifications and generated source). The third button generates a workload specification, which lists each cursor and container operation that is to be performed with its execution frequency. (This information was collected while instantiating previous tabs). The last button sends this specification to ContainerStore's design wizard to be analyzed and critiqued for its efficiency. (Section 5 elaborates this workload specification and analysis).

3 The P3 Generator

The *Jakarta Tool Suite* (JTS) is a set of domain-independent tools for building extensible domain-specific languages and GenVoca (component-based) generators [9]. JTS is written in Jak, an extensible superset of the Java language. Jak minimally extends Java with the addition of metaprogramming features (e.g., syntax tree constructors) so that Java programs can create and manipulate other Java programs. JTS is itself a GenVoca generator, where variants of Jak are assembled from components. One component in the JTS library encapsulates the P3 generator and the P3 component library.

Like its predecessor P2, P3 allows users to specify containers and cursors declaratively, just like database systems provide declarative specifications of relations and relational queries. In particular, P3 adds cursor and container data structure declarations to Java. Examples of these declarations are listed on the lefthand side of Fig. 4; the generated Java code is shown on the right. When Jak parses each of these declarations, it generates the

```
(1) container< emp > empcont;   - — — — — — →  interface empcont { ... }

(2) cursor< empcont > empcursor;  - — — — — — →  interface empcursor { ... }

(3) container ec implements empcont   - — — — — →  class ec implements empcont { ... }
       using odlist( age, malloc( ) );

(4) cursor all(ec e);     - — — — — — — — — →  class all implements empcursor {
                                                  all(ec e) { ... }
                                                  ... }

(5) cursor few(ec e)      - — — — — — — — — →  class few implements empcursor {
       where name() == "Don" && age() > 20          few(ec e) { ... }
       orderby age;                                 ... }
```

Fig. 4. P3 declarations and generated classes.

TABLE 1
P3 Data Structure Components

DS Component	Semantics
`malloc`	elements are stored in transient memory
`persistent`	elements are stored in persistent memory
`odlist( key x, DS y )`	elements are stored on an **x**-ordered doubly-linked list#
`dlist( DS y )`	elements are stored on an unordered doubly-linked list#
`rbtree( key x, DS y )`	elements are stored on a red-black tree with key **x**#
`predindex( predicate p, DS y )`	elements that satisfy predicate **p** are linked on a separate doubly-ordered linked list#
`hashcmp( key x, DS y )`	equality predicates on key **x** are hashed to improve performance#
`hash( key x, int n, DS y )`	elements are hashed on **x** and stored in a hash table with **n** buckets; each bucket implemented by a doubly-linked list#
`bstree( key x, DS y )`	elements are stored on a binary tree with key **x**#

*Note: parameter **y** defines a stack of components that lie below the given component in a P3 type equation.*

appropriate Java interface or class definition and replaces the declaration with the generated code.

Suppose instances of class **emp** are to be stored in a container. Lines (1) and (2) in Fig. 4 concisely declare Java interfaces for containers (**empcont**) and cursors (**empcursor**) that are specialized for **emp** instances. Note that the C++-like syntax for these declarations was chosen deliberately to indicate that container interfaces are parameterized by the elements (**emp**) to be stored and cursor interfaces are parameterized by the container (**empcont**) over which cursor instances will range.

Among the methods in the **empcont** interface (not shown in Fig. 4) are **emp** instance insertion and a test for container overflow. Among the methods in the **empcursor** interface are positioning a cursor on the first **emp** instance of a container, advancing to the next **emp** instance, testing for end-of-container, and get and set methods for each attribute of **emp**.

Each container class is declared separately. Statement (3) defines a container class **ec** that implements the **empcont** interface by storing **emp** instances in an age-attribute-ordered doubly linked list in transient memory. **odlist(age,malloc())** is its P3 type equation that defines both the stacking of components (i.e., **odlist** sits atop of **malloc**) and annotations (i.e., **age** is the key of the **odlist** data structure). Table 1 lists the library of components that P3 currently offers.

It is worth noting that P3 components are *not* the same as traditional parameterized components for data structures (e.g., STL [27]), but instead are "refinements" [33], [5], [6]. To see the difference, consider the interpretation of the composition of hash and bintree:

hash[keyA, 400, bintree[keyB, ...]]

This composition does *not* mean that a container of elements is hash-partitioned into **400** buckets on **keyA** and then each bucket is implemented as a binary tree ordered on **keyB**. The correct interpretation is much closer to the way database systems compose access methods: Each container element is simultaneously linked onto two distinct and independently traversable data structures—a hash structure on **keyA** and a binary tree on **keyB**. Thus, one can access every element of the container by traversing either data structure, where, obviously, the hash structure provides faster access to elements on **keyA**, while the binary tree provides faster access via **keyB**. More generally, P3 allows elements to be linked into *any* number of distinct data structures, thus enabling P3 users to define arbitrarily complicated structures that simply could not be created using traditional parameterized components. This distinction is explained in greater detail in [33], [5, [6].

Each P3 cursor class is also declared separately. Statement (4) declares a cursor class **all** of whose instances return every element of an **ec** container. The syntax of the statement defines the arguments of the constructor of the generated class (i.e., each **all** instance is bound to a particular **ec** container instance). P3 infers that **all** implements the **empcursor** interface (because **ec** containers store **emp** instances and cursors over **ec** containers implement the **empcursor** interface).

Statement (5) declares another cursor class **few** whose instances return in attribute **age** order only those elements of an **ec** container where **name == "Don"** and **age > 20**. (Again, each instance of few is bound to a single **ec** container and only returns instances of that container that satisfy the **few** predicate). Other features of P3 that are not shown in Fig. 4 include parameterized selections (i.e., **city() == x**, where **x** is specified at run-time) and declarations of cursor usage (e.g., retrieval only, element modification/deletion) for optimizing generated code.

TABLE 2
Code Size of Dictionary Benchmark Programs (in Words)

	dlist	bstree	rbtree	hash
JDK	541	N/A	N/A	506
CAL	540	561	561	562
JGL	534	N/A	N/A	540
PIZZA	565	N/A	N/A	509
P3	570	572	572	574

TABLE 3
Execution Times of Dictionary Benchmark Programs (in secs)

	dlist	bstree	rbtree	hash
JDK	82.5*	N/A	N/A	8.2
CAL	117.4	19.4	17.3	13.5**
JGL	116.9	N/A	N/A	8.1
PIZZA	99.2 ***	N/A	N/A	8.7
P3	74.9	13.8	12.8	7.9

**JDK Vector data structure is used here.*
*** CAL does not have explicit support for hash tables; the Set container is used instead. Internally, CAL implements Set by hash table.*
**** Pizza Vector data is used here.*

4 Performance of P3-Generated Code

Generators, contrary to handwritten component libraries, offer a scalable way to produce customized software [5], [11]. The declarative way in which P3 users specify container and cursor implementations through component composition leads to huge families of customized data structures. As shown in [8], significant increases in productivity and major cost reductions both in maintenance and experimentation with different container implementations can result from generators. While we have not yet used P3 in a sophisticated Java application, we have performed preliminary benchmarks on P3-generated code to assure us that P3 is on a trajectory that is comparable with its predecessors [33], [5], [6]. In this section, we review some of our preliminary results on P3's performance.

The *Container and Algorithm Library* (CAL) [39] and the *Java Generic Collection Library* (JGL) [18] are two popular and publicly available Java data structure libraries. Both are based on STL [27] and are optimized for performance. *Pizza* (a dialect of Java that supports parametric polymorphism [29]) and Sun's *Java Development Kit* (JDK) also provide simple data structures, so we also included them in our study. Presently, CAL and JGL support features (adaptors for stacks, queues, etc.) that P3 does not yet offer. (Adaptors can be encapsulated as P3 components that will be stacked on top of P3 containers to give them noncontainer interfaces; so, there is no a priori reason why such capabilities/componentry cannot eventually be added to P3).

We performed a number of experiments that benchmarked productivity and performance; the most revealing of which are presented in Tables 2 and 3. The benchmark of [5] was used to evaluate the performance of the Booch Components, libg++, and the P1 and P2 generators. We used this program for our studies. The program spell-checks a document against a dictionary of 25,000 words. The main activities are inserting randomly ordered words of the dictionary into a container, inserting words of the target document into a second container and eliminating duplicates, and printing those words of the document container that do not appear in the dictionary. The document that we used was the Declaration of Independence (~1,600 words).

We used JDK, CAL, JGL, Pizza, and P3 to implement this program using four different container implementations: doubly linked lists, binary search trees, red-black trees, and hash tables. The benchmarks were executed on a Pentium Pro 200 with 64 MB of memory, running Windows NT Workstation version 4.0. The programs were compiled and executed using JDK version 1.1.3, with the **-O** optimization option. We also recompiled the CAL beta 2 and JGL 2.0.2 libraries using JDK version 1.1.3 to ensure the validity of comparison.

Table 2 shows the program sizes for different libraries. (Sizes were obtained by removing comments and using the Unix **wc** utility to count the words). P3 programs are slightly longer than the corresponding CAL, JGL, Pizza, and JDK programs because P3 declarative specifications are more verbose than class references to Java packages. Such differences are not significant because P3 can generate vast numbers of data structures that have no counterpart in the CAL, JGL, Pizza, and JDK libraries. In such cases, these libraries would not be of much help as the target data structure would have to be written by hand. The brevity of the corresponding P3 programs and the speed at which their Java source is produced would be unchallenged. So too would the ability to alter container implementations quickly and easily (by merely redefining the P3 type equation and recompiling); significantly more work would be needed using Pizza, CAL, JGL, and JDK.

Table 3 lists the execution times for each program. In general, P3 programs outperform their hand-coded counterparts for two reasons. First, both CAL and JGL are based on STL, but, since Java does not support templates, both have to rely extensively on inheritance. This introduces additional dispatches and down-casts, which slows execution. Second, and more significant, there is inherent overhead in the JDK, CAL, Pizza, and JGL designs. These libraries are designed for *generic* applications, whereas the programs generated by P3 are produced for a *specific* task. Consider element comparisons. P3 directly inlines comparison expressions, whereas CAL and JGL programs have to use a "predicate" object that encapsulates a function to evaluate that predicate on a given element. (This is a common way to work around the lack of function pointers in Java.) Note that this function can*not* be optimized by the Java compiler

because it knows nothing about query optimization. There are other inefficiencies that preclude significant optimizations that generators can provide.

The point of our experiments was to provide minimal confirmation that P3 generates code comparable to that written by hand. Clearly, many more experiments are needed. We have no illusions that this simple example is sufficient in any way; our goal at this stage of our research is to demonstrate that the performance of P3-generated code conforms to that observed in earlier generators, which it does.

5 The P3 Design Wizard

A fundamental problem in all component-based generators is: Given a workload specification and a set of components, how should one select and assemble components to define an appropriate application implementation? In the case of P3, what type equation (data structure) would efficiently process a given workload? This is a difficult problem for two reasons.

First, software designers are rarely aware of the actual workload that an application will subject a data structure. A designer will know the kinds of queries asked (e.g., since these queries will be specified as **cursor** declarations), but the actual frequency with which particular **cursor** classes are instantiated and elements are retrieved will not be known until run-time. At best, only educated guesses can be made (and, often, these estimates are determined instinctively).

Second, even if a workload is known precisely, it can be a challenging problem to determine an efficient data structure. When a workload is simple, the problem is easy. For example, if elements of a container are to be accessed only via the predicate **N == <value>**, then a hash table with elements hashed on field **N** is likely to be an optimal choice. However, if workloads become slightly more complicated, it is hard to tell what data structure would be best. For example, if there are 20,000 elements, 3,000 elements are inserted and deleted per time period, fields **S** and **N** are updated 1,000 times per period, elements are retrieved using predicate **N == <value> && A=="b"** 2,000 times per period, and all elements are retrieved in **S** order 50 times per period, what data structure would most efficiently support this workload? The answer is not obvious even to experienced programmers.

To solve the first problem, one can instrument generated code so that it collects workload statistics at run-time. So, initially, one fields an application knowing full well that its data structures are not optimal. After a period of time, enough statistics will have been collected so that a more appropriate data structure can be determined. The application **cursor** and **container** classes are then regenerated and the old classes discarded. A new cycle of collect-statistics-and-regenerate then begins. Normally, a programmer is in the loop to close the cycle (i.e., a programmer decides how long to collect statistics, how to use these statistics to deduce a better data structure, and when to initiate the class regeneration and replacement). However, this loop could be closed *without* programmer intervention. That is, the *application* determines when enough statistics have been collected, a tool called a *design wizard* finds a more efficient data structure given this workload, and if regeneration is warranted, class regeneration and replacement is performed automatically. Such software is called *self-adaptive* [23], [36], [30], [2] and may be the ultimate way to minimize software development and maintenance costs through component reuse.

The key to achieving self-adaptive software requires a solution to the second problem—deducing an efficient type equation for given a workload. This requires a kind of knowledge that is not present in GenVoca domain models (and domain models in general). Knowledge of when and how to use a component effectively to maximize performance or to meet a design objective is quite different than that of design rules (i.e., requirements that define the correct usage of a component [7]). What form this knowledge will take, what is a general model to express such knowledge, and how to optimize type equations remain open problems. Short of proposing a general-purpose theory, it is possible to develop ad hoc techniques for given domains and, in particular, for data structures. By abstracting from specific solutions in different domains (i.e., performing a "domain analysis" on these solutions), a general theory may result.

For now, however, we outline an approach that we have found effective to optimize and critique P3 type equations automatically given a workload specification. While the solution itself is domain-specific, it does constitute a valuable first step toward self-adaptive software and a general model of design wizards.

5.1 P3 Workload Specifications

Data structure optimization is a well-studied problem. Because P3 presents a relational-like interface to data structures, relational database optimization models are an obvious starting point (e.g., [26]). A *workload* on a database relation (or P3 container) is characterized by the type and cardinality of individual attributes of an element, plus the frequency with which each container or cursor operation is performed. Fig. 5 illustrates a workload specification file produced by the ContainerStore applet. (The information of Fig. 5 was provided by a ContainerStore client when he/she filled in the operation frequency slots in the ContainerStore specification tabs or it might be the result of a statistical analysis of running instance of the target application.) It states that there are 5,000 elements in a container. Each element has two fields, one is a String called **name** that has 5,000 unique values, etc. Three hundred elements are inserted per time period, all elements are retrieved in **name** order 100 times per period, and so on. The type equation (which implements the container whose workload is defined in Fig. 5) that is to be critiqued is **odlist(age,malloc())**.

5.2 Cost Model

Given a workload W and a container implementation (type equation) T, we want to estimate the cost of processing W using T. This is accomplished by synthesizing cost functions.[2] The cost function we seek, $Cost(T, W)$, is the sum of

2. The method by which cost functions are produced is exactly the same as that used in P3 code synthesis and type equation explanations.

```
workload {
     cardinality = 5000;

     attributes {
          #id       type      cardinality
          #--------------------------
          name      String    5000;
          age       int       60;
     }

     work {
          #operation          frequency
          #--------------------------
          insertion           300;
          deletion            300;
          ret orderby name    100;
          ret where name() == "Don" &&
              age() > 20
              orderby age     100;
     }

     Equation = odlist(age, malloc());
}
```

Fig. 5. P3 workload specification.

the costs of processing each individual cursor and container operation times its execution frequency. The cost of an individual operation is the sum of the costs contributed by individual layers of T. For example, every layer performs some action when an element is inserted into a container. Thus, the cost of an element insertion is equal to the sum of the costs of insertion actions that are performed by each layer (see Fig. 6). The same holds for attribute update and element deletion. Retrieval costs are estimated a bit differently, as query optimization is involved. A retrieval predicate is processed by traversing a single data structure. The data structure that is to be traversed (i.e., the structure whose traversal algorithms are to be generated) is the one that returns the minimum cost estimate for processing that predicate. This "polling" of layers/data structures by P3 is called *query optimization*. Selected functions that define $Cost(T, W)$ are summarized in Table 4, where n denotes the number of elements in a container, b the number of hash buckets, and c is a constant. Different data structures will have different values for c for different operations, where particular values are determined by benchmarking P3 data structures on a specific platform.[3,4]

3. *Equality retrieval* are predicates of the form **key == value**; *range retrieval* are predicates of the form **low-value < key < high-value**; *scan retrievals* do not qualify elements on key values.

4. Although we list only the higher-order terms in Table 4, we included lower-order terms in our prototype. It turns out that basic problems which plague database researchers for obtaining accurate estimates of query processing costs also hinder us (see [31]). For example, it is well-known that accurate estimates of query and subquery selectivity are difficult to obtain. While we could use more advanced techniques of estimation, experience has shown that this is overkill for typical P3 applications. When designing data structures, most programmers do not sit down with a calculator to determine the most efficient data structure; rather, they apply heuristics learned in their data structure courses to design and implement their container. These heuristics are captured by these equations when n, the number of elements in a container, is "sufficiently large." When the number of elements is not known (which is generally the case), these heuristics are reasonable. See [5], [6] for examples.

$$Cost(T, W) = I(T) \times InsFreq + D(T) \times DelFreq + \sum_j (U(T, Field_j) \times UpdFreq_j) + \sum_j (R(T, Ret_j) \times RetFreq_j)$$

$$I(T) = \sum_{i \in T} insertionCost(layer_i)$$

$$D(T) = \sum_{i \in T} deletionCost(layer_i)$$

$$U(T, Field_j) = \sum_{i \in T} updateCost(layer_i, Field_j)$$

$$R(T, Ret_j) = Min_{i \in T}(retrieval(layer_i, Ret_j))$$

Fig. 6. P3 cost model.

$Cost(T, W)$ again is used to evaluate a particular design T for a workload W. Ideally, a design wizard must walk the space of all legal type equations and find the equation T that minimizes $Cost(T, W)$. In the next section, we explain how this space is defined and, later, how our wizard walks this space.

5.3 The Space of P3 Type Equations

P3 components are characterized by three kinds of attributes: properties, signatures, and design rules. Together they define the space of all syntactically and semantically correct P3 type equations. A *Layer Declaration File* (LDF) is a specification of this information, an example of which is shown in Fig. 7.

Properties are attributes that classify components [7]. In Fig. 7, six different properties are defined. **logical_key** is the propositional symbol for the attribute that defines "a key-ordered component," i.e., a P3 component that implements a data structure that stores elements in key order. Red-black trees and ordered doubly linked lists have this property. Similarly, **hash_key** is the propositional symbol for the attribute that defines "a hash component," i.e., a P3 component that implements a data structure that stores elements via hashing. As we will see shortly, properties are used to express both design rules and type equation rewrite rules (discussed in the next section). Consistent with the experience discussed in [7], determining these properties is a fairly straightforward task.

Signatures define the export and import interfaces of a component; these properties are used to determine if a component usage in a type equation is syntactically correct. In Fig. 7, **ds = {...}** denotes the usual GenVoca syntax for a *realm* (i.e., library) of components that implement the interface **ds** [Bat92]. Three such components are listed: **rbtree**, **delflag**, and **malloc**. The signature of the **rbtree** (red-black) tree is circled. **rbtree** has a **keyfield** parameter and **ds** parameter (which means that **rbtree** can be composed with other **ds** components). In contrast, the **malloc** component has no parameters.

Not all syntactically correct type equations are semantically correct. Domain-specific constraints called *design rules* are needed to define the legal uses of a component. The algorithms that we use for design rule checking are given in [7]. Design rules are expressed in two parts. First, properties that are asserted or negated by a component are broadcast

TABLE 4
Selected Individual Cost Equations

Layers	insertion	deletion	update	equality retrieval	range retrieval	scan retrieval
dlist	c	c	c	$c*n$	$c*n$	$c*n$
rbtree	$c*log(n)$	$c*log(n)$	*key:* $c*log(n)$ *non-key:* c	*key:* $c*log(n)$ *non-key:* $c*n$	*key:* $c*log(n)$ *non-key:* $c*n$	$c*n$
hash	c	c	*key:* c *non-key:* c	*key:* $c(n/b)$ *non-key:* $c*n$	$c*n$	$c*n$

to all layers that lie above it and below it in a type equation. These properties are declared by the **asserted properties** and **negated properties** statements. For example, the **malloc** component broadcasts the asserted property **transmem** when it is used in a type equation. Similarly, the **rbtree** component broadcasts the asserted properties **retrieval** and **logical_key**.

Second, preconditions for component usage are expressed as conjunctive predicates. Asserted properties are expressed with the **require** statement; negated properties with the **forbid** statement. Thus, if a component **X** has the declarations:

require above = { A, B }
forbid above = { C }

they define the predicate **A** $\wedge$ **B** $\wedge$ $\neg$**C** which must be satisfied by layers that lie *above* **X** in a type equation. By replacing "**above**" with "**below**," the predicate must be satisfied by layers that lie *below* **X** in a type equation. (Thus, different conditions can be imposed on layers above **X** and below **X** in an equation). As an example that combines both property broadcasting with preconditions, the **delflag** layer in Fig. 7 allows only one instance of itself in a type equation. That is, the first **delflag** instance will broadcast the **delete** property, while a second **delflag** will detect its presence when its precondition $\neg$**delete** fails.

5.4 Automatic Optimization of Equations

The space of all P3 type equations is the set of all design-rule-correct type equations that can be composed using the given components. The size of this space is enormous: If there are k components in the P3 library, the number of type equations with c components is $O(k^c)$. So, even for small k and c, an exhaustive search is infeasible. While the number of components in an equation is theoretically unbounded, we know from experience that domain experts can quickly identify an efficient equation with few (i.e., typically under 10) components.

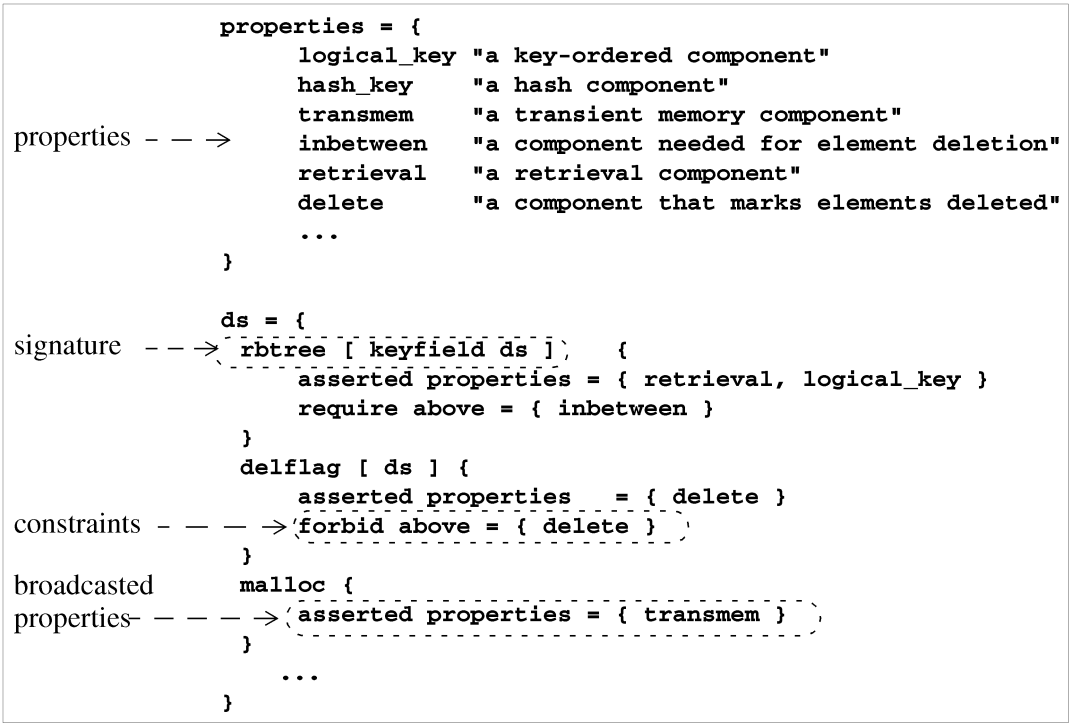

Fig. 7. A P3 layer declaration file.

The number of equations that are relevant to an application is a subspace of the entire P3 space. One way in which this subspace can be generated is by applying rewrite rules that transform one equation into an equivalent equation, starting from an initial feasible solution/equation (e.g., [17]). The rewrite rules that we use are derived from an analysis of the heuristics that we have personally applied to produce efficient type equations manually. In particular, our intuitive optimization strategy has been guided by three heuristics:

- When an equation rewrite is attempted, we check that the resulting equation is *consistent* (i.e., it is syntactically correct and it satisfies the design rules);
- The cost of the rewritten equation is unchanged or lowered;
- Rewrites are considered in an order (we feel) will most likely lead to a new equation with lower cost.

Some of our rules deal with element attributes. Consider the following rewrite that is expressed in two parts:

1. If an element attribute **A** is listed as an **orderby** key in the workload specification, then try to insert a **logical_key** layer (such as a red-black tree or an ordered-list) with **A** as its key.

The idea of this rewrite is that it is cheaper to store elements in sorted order rather than sorting an unordered set of elements on demand. This rewrite may fail if there already exists a **logical_key** layer with that attribute as key. (The reason for failure is that the $Cost(T, W)$ of the rewritten equation T will be higher—the rationale is that a single data structure that maintains element order is usually cheaper than two structures maintaining the same order.) This leads to the second part of the rewrite:

2. If 1 fails, then try to replace the **logical_key** layer with **A** as its key with a more efficient **logical_key** layer.

The idea of this rewrite is that if there already exists a data structure that maintains elements in key order, there may be a more efficient data structure to accomplish the same task. This rewrite attempts to find a such a replacement.

Readers may have observed the use of Layer Declaration File properties in expressing rules. To apply the above rule, our design wizard searches its library for components that assert the **logical_key** property. These components are candidates for insertion or replacement in the above rule. Different rules qualify components on different properties. Consider a second rewrite:

- If element attribute **A** is used in an equality retrieval predicate (e.g., **name == "Don"**), then try to insert a **hash_key** component with **A** as its key; if there already exists such a layer, try to substitute it with a more efficient **hash_key** layer.[5]

These and similar rules are growth rules—i.e., they add components to type equations. There are growth rules that do no involve element attributes. There are also *shrink rules*—i.e., rules that remove components from type equations. An example is:

- Remove a component from a type equation if it increases *Cost*.

The optimization of a P3 type equation is similar to an AI planning process [16]. We discovered that optimizing P3 equations manually followed a best-first (greedy) heuristic; we automated this search to find a correct and efficient type equation with regard to the given workload, cost models, and layer declarations. Because the equations that are retained in the search have progressively lower cost, we are guaranteed to find a local minimum. The search can begin from scratch, starting from a trivial data structure—such as a doubly linked list in transient memory. However, when used with the ContainerStore applet, the search begins with the type equation that was specified in the workload.

Overall, we have about 10 different rewrite rules. The basic algorithm that we use to apply these rewrites to optimize type equations is:

```
for each element attribute A {
   apply each "attribute growth"
     rewrite for A;
}
apply each "nonattribute growth" rewrite;
apply each "shrink" rewrite;
```

The algorithm is run to a fixpoint (i.e., the algorithm is continually invoked until no further rewriting is possible) and, thus, will identify a local minimum. If the P3 subspace defined by our rewrite rules is well-formed (i.e., has only one minimum), our algorithm is guaranteed to find it. If the subspace has several local minima, our algorithm will locate one, but not necessarily the global minimum. We are unaware of any theoretical result that would tell us whether a P3 subspace is (or is not) well-formed. Lacking such information, it is possible that a more powerful search algorithm might uncover better results. However, the results we have obtained using this algorithm have been quite good—occasionally better, but never worse than, what we would have manually selected. Moreover, we are unaware of a proposed equation that we could subsequently improve. Although much more work (e.g., using more powerful search algorithms) remains, we believe that a greedy search algorithm is a reasonable first step.[6]

5.5 Critique and Optimization

Given a workload specification, the P3 design wizard applies its rewrites to the input type equation. If there is

5. At present, P3 has only one hash component. A component for dynamic hashing may be added later.

6. Since the conference publication of this paper, we have a proof that the P3 search space can indeed have multiple local minima and that finding the global minimum is NP-hard. An example that illustrates the problem involves processing every query of a set of queries using some keyed data structure (e.g., binary-tree). Suppose that creating a keyed data structure on field A, then another on field B, and a third for field C will allow all queries in the set to be processed efficiently by traversing one of these structures. Call these structures the ABC indexing set. Now, suppose that we had created a keyed structure on field E and a second on field F and all queries of the set could be processed efficiently by traversing either the E structure or F structure. Call this the EF indexing set. Clearly, the ABC indexing set and the EF indexing set are local minima. If our design wizard selects the ABC as an answer, it will be unable to "backtrack" to find solution EF (or vice versa) and, hence, will not find the global minimum.

```
Original Type Equation is:
        odlist(age,
         malloc( ))
cost = 19593

Type Equation we recommend is :
        hashcmp(name,
         hash(name,5000,
          odlist(name,
           malloc( ))))
cost = 1606

Projected improvement: 1119%

Reasons why we choose this type equation:
     hashcmp: field name is hashed because it
          will be faster to compare the values
          of two string fields when they are
          hashed.
     hash: A hash data structure with hash key
          name is used because 11% of the
          operations involve equality retrieval
          on name.
     odlist: A doubly linked list ordered by
          name is used because many retrievals
          will be ordered by name.
```

Fig. 8. Critique and optimization of a Type Equation.

no substantial improvement, the wizard simply reports that no changes to the equation need to be made. A more likely response is that it will have discovered an implementation/equation that has better performance characteristics. Fig. 8 shows the output of a critique using the workload of Fig. 5.

Both the input and revised equations are presented, along with their cost (i.e., $Cost(T, W)$) estimates. An explanation is also presented which provides reasons why the generated equation is better. The reason is that the original data structure linked elements together onto an **age**-ordered list. The workload, on the other hand, demands that all elements of the container be periodically retrieved in **name** order and that individual elements (whose name is "**Don**") be retrieved frequently. The original data structure does not efficiently support this workload at all. The recommended data structure allows elements to be accessed quickly (via hashing) on **name** and that elements be stored in order on **name** (via an ordered linked list). Furthermore, to speed up the search for elements on **name**, the **hashcmp** component is used. (**hashcmp** transforms equality predicates on strings (**name == "Don"**) to include integer comparisons (**hash_of_name == hash("Don") && name == "Don"**). The idea is that integer comparisons are much faster than string comparisons). While most programmers would not think to add this enhancement (probably because it is tedious for a programmer to add by hand), it is quite simple for P3 to do it. The performance enhancements for altering the type equation are predicted by the design wizard to pay-off handsomely. (The actual percentage reported in Fig. 8 is not particularly important; rather, it gives users some idea of how much better the suggested design would be.)

There are two general contributions that design wizards make to automated software development. First, not all users of a generator will be domain experts. Even if they have familiarity with a domain, they may not know as much as an expert, or, in the case of design wizards, a host of domain experts. Design wizards will help avoid blunders and will help users find more efficient implementations for their target systems. Second, and possibly more significant, type equation synthesis is a prerequisite to adaptive software—applications that dynamically change their configuration as a function of current workload. For most domains—including data structures—manual reconfigurations are rarely done because of the costs involved. (The problem becomes even more complicated if data structures are persistent; updating data structures requires the additional cost of unloading data from old structures and reinserting it into the new structures). As a consequence, application users must suffer with degraded performance and application developers must endure the costs of program maintenance. Design wizards have the potential to change this situation dramatically.

6 Related Work

It is widely believed that *domain-specific languages* (DSLs) will significantly impact future software development. DSLs offer concise ways of expressing complex, domain-specific concepts and applications, which in turn can offer substantially reduced maintenance costs, more evolvable software, and significant increases in software productivity [5], [22], [15]. Generators are compilers for DSLs [34]. Component-based generators, such as P2 and P3, show how reusable components form the basis of a powerful technology for producing high-performance, customized applications in a DSL setting (see also [28]).

The automatic selection of data structures is an example of automatic programming [3]. SETL is a set-oriented language where implementations of sets can be specified manually or determined automatically [32]. SETL offers very few set implementations (bit vector, indexed set, and hashing) and relies on a static analysis of an SETL program using heuristics rather than using cost-based optimizations to decide which set implementation to use. AP5 relies more on user-supplied annotations for data structure selection [13].

Deductive program synthesis is another way to achieve automatic programming [3], [35], [25], [24], [19]. The idea is to define a domain theory (typically in first order logic) that expresses fundamental relationships among basic domain entities. A domain theory, together with a theorem prover and theorem-proving tactics, can find a constructive proof for a program specification and extract from this proof computational methods from which a program can be synthesized. Finding a proof may be fully automatic, but frequently requires guidance from users to help navigate through the space of possible proofs. Our design wizard is very different. First, finding a "proof" (a P3 type equation) for a workload specification is trivial—simply implement every container as a doubly-linked list. All container and cursor operations will be processed, but not efficiently. The challenge is finding a P3 type equation that efficiently processes that workload. Second, work on program synthesis has largely focused on generating *algorithms* (e.g., algorithms for solving PDEs [19], algorithms for scheduling

[35], algorithms for computing solar incidence angles [24], etc.); subroutines are the components from which generated algorithms are built. The inferences needed for algorithm synthesis tend to be quite sophisticated (thus requiring theorem provers) because there are very complex relationships among domain entities. In contrast, GenVoca is different both in component scale and in the simplicity of the relationships among domain entities. GenVoca components are *subsystems*—suites of interrelated OO classes. (A P3 component, for example, encapsulates three classes: a cursor class, a container class, and an element class.) As noted in [7], scaling the size of components *and* designing components to be plug-compatible has a nonobvious effect: The relationships that exist among components tend to be very simple, and elementary inferencing (i.e., no theorem provers) is adequate.

Our concept of design wizards resonates with recently proposed notions of *open implementations* (OI) [20] and *Aspect Oriented Programming (AOP)* [21]. Aspects in AOP are very similar to components in GenVoca; they encapsulate changes to be made to multiple classes when an aspect (or feature) is added to an application. Aspect weavers are functions which take a program as input and produce another (more detailed, extended, or refined) program as output. P3 components have an almost identical description. The primary difference is that AOP starts with existing application source, whereas GenVoca decomposes applications into primitive layers and reexpresses them as a compositions of these layers. P3 relies on general results from GenVoca that address the issues of optimizing across multiple layers/aspects and the order in which components/aspects can be legally composed. To our knowledge, there are no corresponding general results for AOP.

The idea of OI is that, when interfaces largely hide implementation details, it should be possible for clients to annotate abstract declarations with profiling (or other implementation-specific) information so that a compiler or server can automatically select the most appropriate implementation that is available. The OI guidelines address design issues, but implementation details are not discussed. When the space of implementations is restricted to a small handful of choices, the solution is straightforward [3], [32], [13]. However, our experience with P3 shows that declarative specifications can map to vast numbers of implementations. While design issues are indeed important, additional difficult problems remain:

1. creating a model that defines the space of possible implementations,
2. using the model to produce efficient implementations,
3. the ability to rank individual implementations in this space, and
4. efficiently walking the space.

GenVoca and design wizard provide a systematic way to address all of these concerns.

The techniques we used for optimizing type equations are very similar to those of rule-based query optimization [14], [37]. A query is represented by an expression where terms correspond to relational operators (e.g., join, sort, select). Query optimization progressively rewrites a query expression according to a set of rules, where the goal is to find the expression with the lowest cost. Since we model data structures as expressions and our design wizard progressively rewrites expressions until no further rewriting produces a more efficient expression, the problems seem identical. However, there are differences. First, constraints among relational operators can be expressed simply by algebraic rewrite rules. In contrast, we do not yet have an algebraic representation for our rules. (In fact, the implementation of our "rules" is pure Java code). Moreover, the correct usage of layers requires design rule checking, which we also have been unable to express as algebraic rewrites. Second, query optimization deals with a rather small set of operators (e.g., join, sort, select), whereas type equation optimization potentially may deal with a much larger set of operators (i.e., tens or hundreds of layers). For these reasons, type equation optimization may be more difficult than query optimization.

7 Conclusions

P3 is a GenVoca generator for container data structures. Although its basic technology was developed earlier [5], [6], P3's novelty is that it has been implemented as a modular extension to the Java language that introduces data-structure-specific statements. These statements enable P3 users to compactly and declaratively specify a family of data structures whose size dwarfs that of hand-coded Java libraries (e.g., CAL, JGL, JDK, Pizza). Besides offering broader coverage, P3 is additionally attractive because it generates efficient code. The basic reason for its efficiency—beyond the fact that the generation techniques are powerful—is that P3 produces data structures for a specific application (where all kinds of optimizations can be performed), whereas conventional libraries only offer generic data structures (where these optimizations have not been applied).

The P3 generator, however, is not sufficient for a practical software development environment. In this paper, we presented the ContainerStore applet, a visual domain-specific programming language that integrates an important suite of tools and services that P3 alone does not provide.[7] The particular services that we discussed are: English-generated explanations of P3 component compositions (which are important as P3 novices will not be familiar with component semantics), automatic validation of compositions with messages suggesting how to repair errors (if errors are detected), automatic generation of P3 code (so that users can study correct P3 specifications), automatic translation of P3 specifications into Java code (i.e., the P3 generator is called), and the automatic critique and optimization of a user-defined P3 component composition given a workload specification (i.e., the P3 design wizard is called).

Among all these services, our design wizard is the most novel. Although its optimization strategies and component rewrite rules are indeed specific to the domain of container data structures, we believe it is the first example of a much more general technology for automatic component selection and composition. The idea of optimizing component compositions by applying domain-specific rewrite rules is

7. The capabilities described in this paper were demonstrated at the DARPA EDCS Workshop in Seattle, July 1997.

certainly not limited to container data structures. The reason is that GenVoca provides a general way in which to create vast "product-lines" from components; applications of GenVoca product-lines are expressed as type equations and the improvement of a particular equation/design is always through component replacement, insertion, and removal (i.e., equation rewrite rules).

Our initial success with the P3 design wizard is encouraging. However, it is essential that design wizards for other domains be created. We believe that analyzing design wizards for different domains may lead to a general model for expressing type equation rewrite rules. Such a model may offer a general purpose technology for achieving adaptive software—i.e., software that automatically reconfigures itself upon noticing a change in its usage/workload. Adaptive software may be the ultimate way to minimize software development and maintenance costs through component reuse.

ACKNOWLEDGMENTS

We thank the anonymous referees who made valuable suggestions that improved the paper. We also thank Rich Cardone and Yannis Smaragdakis for their comments on earlier drafts of this paper, and Lane Warshaw for discussions that revealed the connection between index selection and multiple local minima in a P3 search space. Our Web site is http://www.cs.utexas.edu/users/schwartz/. This work was supported in part by Microsoft, Schlumberger, the University of Texas Applied Research Labs, and the US Department of Defense Advanced Research Projects Agency in cooperation with the US Wright Laboratory Avionics Directorate under contract F33615-91C-1788.

REFERENCES

[1] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.
[2] R. Allen, R. Douence, and D. Garlan, "Specifying and Analyzing Dynamic Software Architectures," *Proc. Conf. Fundamental Approaches to Software Eng.,* Mar. 1998.
[3] R. Balzer, "A Fifteen-Year Perspective on Automatic Programming," *IEEE Trans. Software Eng.,* vol. 11, Nov. 1985.
[4] D. Batory, "On the Complexity of the Index Selection Problem," unpublished manuscript, 1978.
[5] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, "Scalable Software Libraries," *Proc. ACM SIGSOFT,* 1993.
[6] D. Batory, J. Thomas, and M. Sirkin, "Reengineering a Complex Application Using a Scalable Data Structure Compiler," *Proc. ACM SIGSOFT,* 1994.
[7] D. Batory and B.J. Geraci, "Validating Component Compositions and Subjectivity in GenVoca Generators," *IEEE Trans. Software Eng.,* vol. 23, no. 2, pp. 67-82, Feb. 1997.
[8] D. Batory, "Intelligent Components and Software Generators," *Proc. Software Quality Inst. Symp. Software Reliability,* Apr. 1997.
[9] D. Batory, B. Lofaso, and Y. Smaragdakis, "JTS: A Tool Suite for Building GenVoca Generators," *Proc. Fifth Int'l Conf. Software Reuse,* pp. 143-155, June 1998.
[10] I. Baxter, "Design Maintenance Systems," *Comm. ACM,* pp. 73-89, Apr. 1992.
[11] T. Biggerstaff, "The Library Scaling Problem and the Limits of Concrete Component Reuse" *Proc. Int'l Conf. Software Reuse,* pp. 102-110, Nov. 1994.
[12] S.V. Browne and J.W. Moore, "Reuse Library Interoperability and the World Wide Web," *Proc. Int'l Conf. Software Eng.,* pp. 684-691, May 1997.
[13] D. Cohen and N. Campbell, "Automating Relational Operations on Data Structures," *IEEE Software,* May 1993.
[14] D. Das and D. Batory, "Prairie: A Rule Specification Framework for Query Optimizers," *Proc. Int'l Conf. Data Eng.,* pp. 201-210, Mar. 1995.
[15] A. Van Duersen and P. Klint, "Little Languages: Little Maintenance?" *Proc. First ACM SIGPLAN Workshop Domain-Specific Languages,* 1997.
[16] C.M. Eastman, "Automated Space Planning," *Artificial Intelligence,* vol. 4, pp. 41-64, 1973.
[17] G. Graefe and D. DeWitt, "The Exodus Optimizer Generator," *Proc. ACM SIGMOD,* 1987.
[18] P. Jenkins, D. Whitmore, G. Glass, and M. Klobe, "JGL: The Generic Collection Library for Java," ObjectSpace Inc., URL: http://www.objectspace.com/jgl/, 1997.
[19] E. Kant et al., "Synthesis of Mathematical Modeling Software," *IEEE Software,* pp. 30-41, May 1993.
[20] G. Kiczales, J. Lampling, C.V. Lopes, C. Maeda, A. Mendhekar, and G. Murphy, "Open Implementation Design Guidelines," *Proc. Int'l Conf. Software Eng.,* 1997.
[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *Proc. European Conf. Object-Oriented Programming '97* pp. 220-242, 1997.
[22] R. Kieburtz et al., "A Software Engineering Experiment in Software Component Generation," *Proc. Int'l Conf. Software Eng.,* 1996.
[23] R. Laddaga, *Self-Adaptive Software Workshop,* Kestrel Inst., July 1997.
[24] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood, "AMPHION: Automatic Programming for Scientific Subroutine Libraries," *Proc. Int'l Symp. Methodologies for Intelligent Systems,* pp. 326-335, Oct. 1994.
[25] Z. Manna and R. Waldinger, "Fundamentals of Deductive Program Synthesis," *IEEE Trans. Software Eng.,* vol. 18, no. 8, pp. 674-704, Aug. 1992.
[26] M.F. Mitoma and K.B. Irani, "Automatic Database Schema Design and Optimization," *Proc. 1975 Very Large Databases Conf.,* pp. 286-321, 1975.
[27] D.R. Musser, A. Saini, and A. Stepanov, *STL Tutorial & Reference Guide: C++ Programming with the Standard Template Library.* Addison-Wesley, 1996.
[28] G. Novak, "Software Reuse by Specialization of Generic Procedures through Views," *IEEE Trans. Software Eng.,* vol. 23, no. 7, pp. 401-417, July 1997.
[29] M. Odersky and P. Wadler, "Pizza into Java: Translating Theory into Practice," *Proc. ACM Principles of Programming Languages,* 1997.
[30] P. Oreizy, N. Medvidovic, and R.N. Taylor, "Architecture-Based Runtime Software Evolution," *Proc. Int'l Conf. Software Eng.,* 1998.
[31] V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita, "Improved Histograms for Selectivity Estimation of Range Predicates," *Proc. ACM SIGMOD 1996,* pp. 294-305, 1996.
[32] E. Schonberg, J.T. Schwartz, and M. Sharir, "An Automatic Technique for Selection of Data Representations in SETL Programs," *ACM Trans. Prog. Languages and Systems,* pp. 126-143, Apr. 1981.
[33] M. Sirkin, D. Batory, and V. Singhal, "Software Components in a Data Structure Precompiler," *Proc. 15th Int'l Conf. Software Eng.,* pp. 437-446, May 1993.
[34] Y. Smaragdakis and D. Batory, "DiSTiL: A Transformation Library for Data Structures," *Proc. USENIX Conf. Domain-Specific Languages,* 1997.
[35] D.R. Smith, "KIDS: A Semiautomatic Program Development System," *IEEE Trans. Software Eng.,* vol. 16, no. 9, pp. 1,024-1,043, Sept. 1990.
[36] J. Sztipanovits, G. Karsai, and T. Bapty, "Self-Adaptive Software for Signal Processing," *Comm. ACM,* pp. 67-73, May 1998.
[37] L. Warshaw, D. Miranker, and T. Wang, "A General Purpose Rule Language as the Basis of a Query Optimizer," UTCS TR97-19, Univ. of Texas at Austin, July 1997.
[38] J.S. Poulin and K.J. Werkman, "Melding Structured Abstracts and the World Wide Web for Retrieval of Reusable Components," *Proc. Symp. Software Reusability,* pp. 160-168, 1995.
[39] X3M Solutions, "CAL: The Container and Algorithm Library for the Java Platform,"URL: http:// www.x3m.com/products/cal/, 1997.

∗ The authors' photographs and biographies are not available at this time.