5th International Conference on Software Reuse (ICSR), Victoria, Canada, June 1998

Web-Advertised Generators and Design Wizards

Don Batory, Gang Chen, Eric Robertson, and Tao Wang
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Domain-specific generators will increasingly rely on Web-based applets for declarative specifications of target applications. Applets will communicate with generators via servers 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 applet, its generator, and design wizard.

1 Introduction

The World Wide Web offers the potential to revolutionize software libraries [Pou95, Bro97]. Instead of purchasing static libraries of limited capabilities, down-loadable applets can provide graphics and text-based domain-specific languages for declaratively specifying target applications [Nov95]. Specifications may involve filling out forms or editing diagrams on one or more HTML pages. Once completed, the contents of these pages can then be up-loaded to a generator (via a server) that automatically manufactures the application source code. The code is then transmitted back to the user (possibly in exchange for billing information). There are numerous advantages to this approach: an almost immediate response to clients requesting customized code; a single copy of a generator can be shared remotely by many users; proprietary generator technologies can be kept safely in-house yet used remotely by customers; the usage and results of generators can be monitored for later tuning and product improvement; and so on.

We are now exploring this approach using 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 [Bat93-94]. 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 makes P3 attractive is that it produces data structure code whose efficiency is comparable (or better than) hand-coded Java class libraries that are currently available.

To advertise and disseminate 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 the 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 (we hope) will make P3 programming easier — *tools and services that P3 alone cannot provide*. The ContainerStore offers the 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 component compositions (or in any other phase of specification), the errors are caught immediately and explanations of how to repair the errors are offered. 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 new composition is reported and reasons are given to explain why this alternative composition should be an improvement. In this way, design wizards can 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 distributed across five tabs, the most interesting of which are shown in Figure 1-2. 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 second tab — called the *Type Equation Tab* (Figure 1) — presents a visual interface for defining customized container implementations as a composition of P3 components (also known as a *P3 type equation*). Compositions are actually stacks of components. Figure 1 shows two stacks: the "**`Equation`**" stack has components **`rbtree`** on top of **`hash`** on top of **`malloc`**; the "**`Equation2`**" stack has **`dlist`** on top of **`hashcmp`** 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 Figure 1, annotations to the hash data structure component (called **`hash`**) are specified by clicking **`hash`** and entering the name of the key to hash (**`age`**) and the number of buckets (**`100`**) in *Annotations* fields.

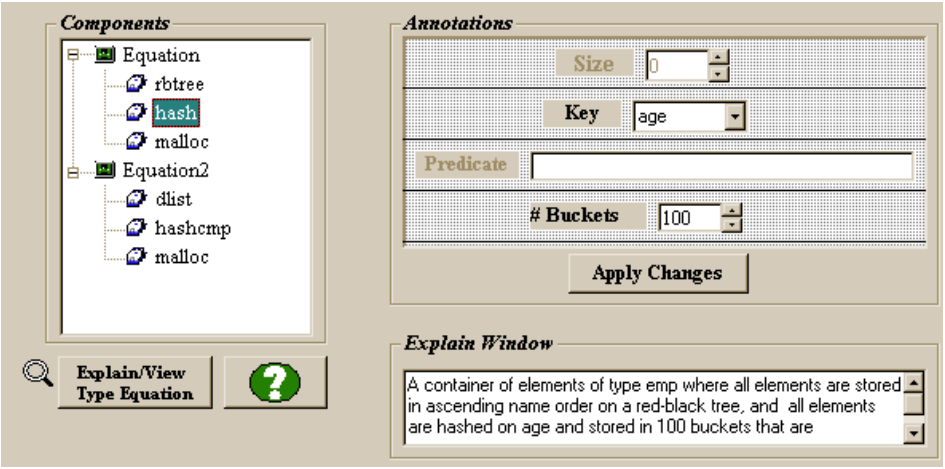

Figure 1. The Type Equation Tab

Once a type equation (component stack) has been constructed, the *Explain/View Type Equation* button is pressed. If the composition of components is valid, an explanation of its meaning is shown in the *Explain Window*. In Figure 1, 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.

This explanation is 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 equa-

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

tion is incorrect — i.e., constraints for the correct usage of a component have been violated (see [Bat97a])— 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 message suggests that the **`hashcmp`** and **`dlist`** layers be transposed (the actual reasons are rather low-level and are not given — only that a correct composition has **`hashcmp`** above **`dlist`**). Applying this rewrite (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 a user specifies the names of the container classes that he/she wishes to build and the type equation that defines its implementation. (Suppose container class named **`ec`** is defined). The fourth tab specifies cursor classes (Figure 2). A *cursor* is a run-time object that is used to reference, update, and delete elements in a container. In Figure 2, the cursor class **`few`** is defined. Its constructor has a single parameter of type **`ec`** — meaning that every **`few`** instance will be bound to an instance of container **`ec`**. The selection predicate is specified incrementally using the *Predicate Builder*, which allows clauses of the form (attribute relation value) to be declared separately. The predicate for **`few`** selects **`ec`** 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.

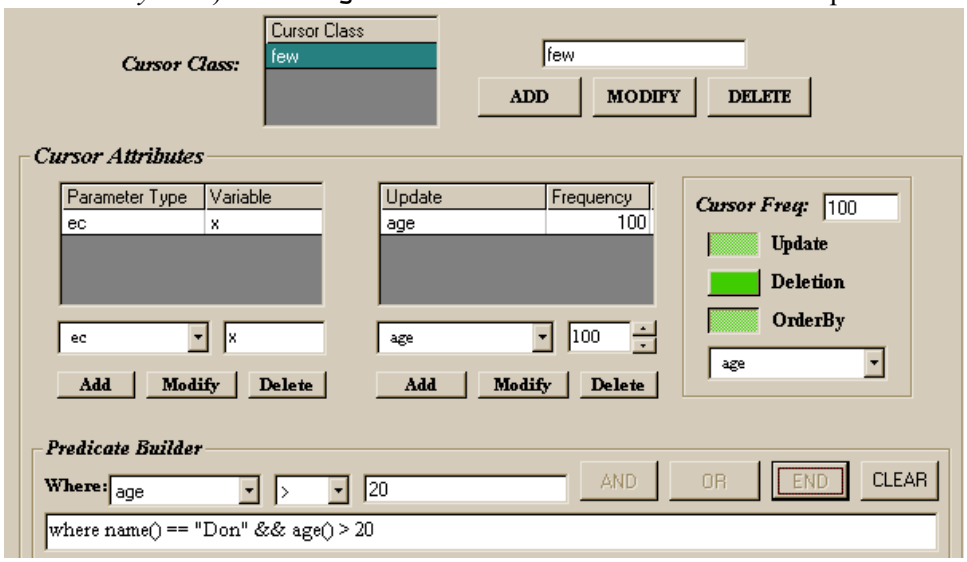

Figure 2. The Cursor Tab

Finally, the Generate tab presents a scrollable window and four buttons. One button generates the P3 language specification of a ContainerStore declaration; a second button sends this specification to a server, where the P3 generator converts this specification into Java source. This source is returned and displayed in the scrollable window. (Section 3 illustrates this source code). 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 in 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 (or component-based) generators [Bat97b]. 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 offers a powerful relational interface to container data structures. In particular, P3 adds cursor and container data structure declarations to Java. Examples of these declarations are listed on the left-hand side of Figure 3; the generated Java code is shown on the right. When Jak parses each of these declarations, it generates the 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) concisely declare Java interfaces for containers (**`empcont`**) and cursors (**`empcurs`**) 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 Figure 3) 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`**.

```
(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;                                ... }
```

Figure 3. P3 Declarations and Generated Classes

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 (realm) of components that P3 currently offers.

Each cursor class is also declared separately. Statement (4) declares a cursor class **`all`** 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

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 stored 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[#]
`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 that component.

Table 1. P3 Data Structure Components

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`. Other features of P3 that are not shown in Figure 3 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, etc.) for optimizing generated code.

4 Performance of P3-Generated Code

Generators, contrary to hand-written component libraries, offer a scalable way to produce customized software [Bat93, Big94]. The declarative way in which P3 users specify container and cursor implementations through component composition leads to scalable families of customized data structures. As shown in [Bat97c], 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.

The *Container and Algorithm Library (CAL)* [X3M97] and the *Java Generic Collection Library (JGL)* [Jer97] are two popular and publicly available Java data structure libraries. Both are based on STL [Ste94] and are optimized for performance. Sun's *Java Development Kit (JDK)* also provides some simple data structures, so we also included it 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 non-container 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 here. The benchmark of [Bat93] was used to evaluate the performance of the Booch Components, libg++, and the P1 and P2 generators. We used this same 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 (~1600 words).

We used JDK, CAL, JGL, 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, and JDK programs because P3 declarative specifications are more verbose than class references in Java packages. Such differences are not significant, because P3 can generate vast numbers of data structures that have no counterpart in the CAL, JGL, 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 CAL, JGL, and JDK.

	dlist	bstree	rbtree	hash
JDK	541	N/A	N/A	506
CAL	540	561	561	562
JGL	534	N/A	N/A	540
P3	568	568	568	568

Table 2. Code size of dictionary benchmark programs (in words)

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 doesn't support templates, both have to rely extensively on inheritance. This introduces many virtual method lookups, which slows execution. Second, there is inherent overhead in the JDK, CAL, and JGL designs. These

	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
P3	74.9	13.8	12.8	7.9

* The Vector data structure provided by JDK 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.

Table 3. Execution times of dictionary benchmark programs (in secs)

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 or 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 compiler (i.e., there is no query optimization) nor inlined because it is virtual. There are other inefficiencies that preclude significant optimizations that generators can provide. Further analysis is given in [Che97].

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? In the case of P3, what type equation (data structure) would most 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 won't be known until run-time. At best, only educated guesses can be made (and often, these estimates are determined through gut instincts).

Second, even if a workload is known precisely, it can be a challenging problem to infer 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, 3000 elements are inserted and deleted per time period, fields **`S`** and **`N`** are updated 1000 times per period, elements are retrieved using predicate **`N == <value> && A=="b"`** 2000 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 isn't 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 inferred. 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 infer 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* infers an optimal data structure given this workload, and if regeneration is warranted, class regeneration and replacement is performed automatically. Such software is called *self-adaptive* [Lad97], and may be the penultimate 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 different kind of knowledge that is not present in GenVoca domain models (and domain models in general). Knowledge of when and how to effectively use a component to maximize performance or to meet a design objective is quite different than design rules

(i.e., requirements that define the correct usage of a component [Bat97a]). 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.

We believe that parameterized programming [Gog86] (of which GenVoca is an example) offers a general framework for modeling systems as equations (see also [Sre97]). Heuristics for improving equations (software designs) may be understood as algebraic rewrites (i.e., replace the component composition `a[b[…]]` with `c[…]` under specific conditions). We are working on a formal model that explores this approach, and will report our results in future papers.

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 [Mit75]. 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. Figure 4 illustrates a workload specification file produced by the ContainerStore applet. (The information in Figure 4 was collected in the various tabs that were filled out by a ContainerStore client). It states that there are 5,000 elements in a container. Each element has two fields, one is a String called **`name`** that has 10,000 unique values, etc. 300 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 Figure 4) that is to be critiqued is **`odlist(age, malloc())`**.

```
workload {
    cardinality = 5000;

    attributes {
        #id       type     cardinality
        #---------------------------------
        name      String   10000;
        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());
}
```

Figure 4. P3 Workload Specification

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 this using cost equations [Mit75]. The cost function we seek, *Cost(T,W)*, is the sum of 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 Figure 5). 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 is the one that returns the minimum cost estimate for processing that predicate. This "polling" of layers/data structures to see which is to be used is called *query optimization*. The general equations that define *Cost(T,W)* are summarized in .Specific equations for layer insertion, deletion, etc. are elementary. Table 4 samples some of the equations used, where *n* denotes the number of elements in a container and *c* is a constant. Different equations have different values for *c*, where particular values are determined by benchmarking P3 software on a specific platform.[2] .

$$\text{Cost}(T, W) = I(T) \times \text{InsFreq} + D(T) \times \text{DelFreq} + \sum_{i \in T} (U(T, \text{Field}_i) \times \text{UpdFreq}_i) + \sum_{i \in T} (R(T, \text{Ret}_i) \times \text{RetFreq}_i$$

$$I(T) = \sum_{i \in T} \text{insertion}(\text{layer}_i)$$

$$D(T) = \sum_{i \in T} \text{deletion}(\text{layer}_i)$$

$$U(T, \text{Field}_j) = \sum_{i \in T} \text{update}(\text{layer}_i, \text{Field}_j)$$

$$R(T, \text{Ret}_j) = \text{Min}_{i \in T}(\text{retrieval}(\text{layer}_i, \text{Ret}_j))$$

Figure 5. P3 Cost Model

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

Table 4. Selected Individual Cost Equations

Cost(T,W), again, is used to evaluate a particular design *T* for a workload *W*. 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 the wizard walks this space.

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

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 Figure 6.

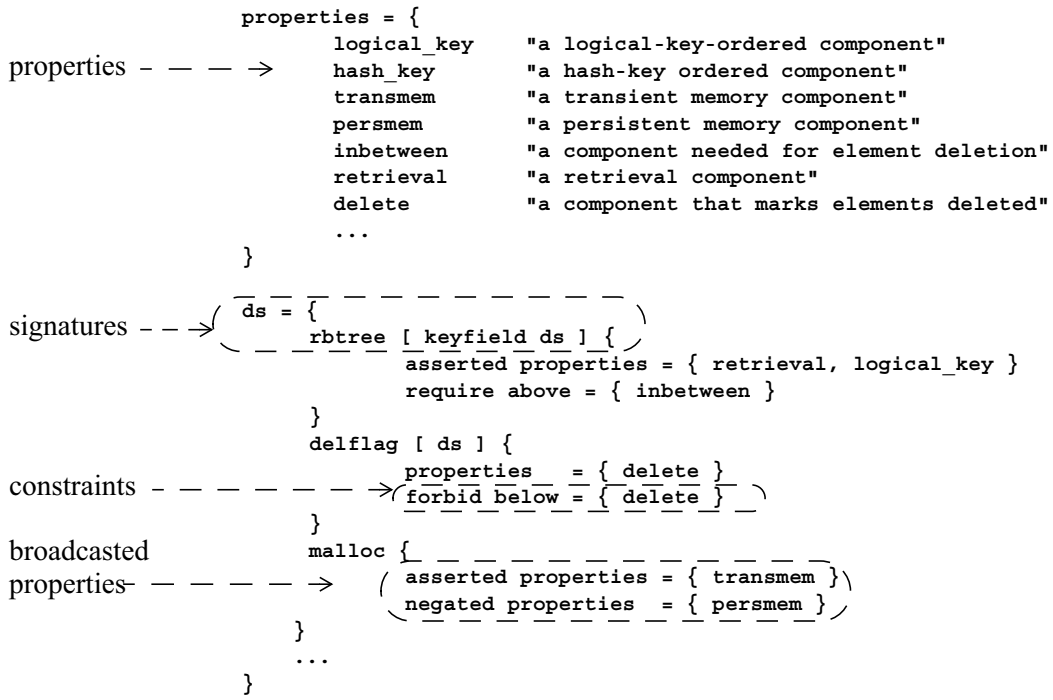

Figure 6. A P3 Layer Declaration File

Properties are attributes that classify components. In Figure 6, seven 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, ordered doubly-linked lists, etc. are components that 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).

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 Figure 6, we circled the signature for red-black trees . The syntax **`ds = { … }`** defines a *realm* (i.e., library) of components that implement the interface **`ds`**. Component is **`rbtree`** exports the **`ds`** interface (because it is a member of realm **`ds`**). It has a **`keyfield`** parameter and parameter **`ds`** (which means that **`rbtree`** can be composed with other **`ds`** components). As another example, the **`malloc`** component exports the **`ds`** interface (because it belongs to realm **`ds`**) and 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 [Bat97a]. Design rules are expressed in two parts. First, properties that are asserted or negated by a component are broadcast 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`** and the negated property

persmem 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**", this 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 Figure 6 allows only one instance of itself in a type equation. That is, the first **delflag** instance will assert the **delete** property, while the second **delflag** will detect its presence when its precondition $\neg$**delete** fails.

5.4 Automatic Optimization of Type Equations

The optimization of a P3 type equation is similar to an AI planning process [Eas73]. We use a best-first (i.e., greedy) search to find a correct type equation with the lowest cost with regard to the given workload, cost models, and layer declarations. The search can begin from scratch (i.e., an equation that has no components). However, when used with the ContainerStore applet, the search begins with the type equation that was specified in the workload.

The space of all P3 type equations is a graph **G(T) = { V,E }**, where **V** is the set of all correct type equations that can be composed using the given components and **E** is the set of edges. There is an edge between type equation *m* and *n*, if and only if *n* can be obtained by inserting, deleting, or replacing a component from *m*. 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. A reasonable start is to use a best-first (greedy) search with the following heuristics:

- when an equation rewrite is attempted, we check that the resulting equation is syntactically correct, that it satisfies design rules, and that its cost is unchanged or lowered.
- rewrites are considered in an order (we feel) will most likely lead to a new equation with lower cost.

The rewrite rules that we use are derived from heuristics that we have applied to produce efficient type equations manually. Some of these rules deal with element attributes. Consider the following rewrite that is expressed in two parts:

- (a) 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 will 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 always cheaper than two structures maintaining the same order). This leads to the second part of the rewrite:

- (b) If (a) 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 LDF 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 will qualify components on different properties. Consider a second rewrite:

- if element attribute **`A`** is used in an equality retrieval predicate (e.g., **`name == "Dan"`**), 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.[3]

These and similar rules are *growth rules* — i.e., they add components to type equations. There are growth rules that do not 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*.

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 "non-attribute growth" rewrite;
apply each "shrink" rewrite;
```

Optionally, the above algorithm can be run to a fixpoint (i.e., the algorithm is continually invoked until no further rewriting is possible). Since our search algorithm is not exhaustive, there is no guarantee that the generated equation has minimal cost. However, it does execute quickly (under a second of CPU time). As we will see in the next section, the algorithm seems to work well.

5.5 Critique and Optimization

Given a workload specification, the P3 design wizard applies its rewrites to the input type equation. If there is 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. Figure 7 shows the output of a critique using the workload of Figure 4. 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. In this particular case, 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 doesn't 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 searching for elements on **`name`**, the **`hashcmp`** component is used. (This component 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 com-

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

parisons). 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.

```
Equation: 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.
```

Figure 7. Critique and Optimization of a Type Equation

There are two general contributions we see that design wizards will 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 building adaptive software systems — systems 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. 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 [Bat93, Kie96, Due97]. Generators are compilers for DSLs [Sma97]. 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.

The importance of the World Wide Web to software reuse has been recognized for some time. Among the more recent efforts are those that apply standards to library catalogs to foster the interoperability of different software repositories [Pou95, Bro97]. The Web provides a convenient (and inexpensive) medium in which to merge the contents of distributed libraries and present a uniform interface for searching and selecting components. Our work is different. While there is nothing new or remarkable about using applets as Web-based DSLs and for ordering/purchasing software via the Web, we believe that the ContainerStore applet is among the first that tie DSLs, component-based software generators, and the web together in a coherent and workable manner. To our knowledge, Novak was the first to have used this combination of

technologies (with data structure generators, no less!) [Nov95]. His generation technology is quite different from ours (e.g., there is no design wizard and the applet is really an X-windows application) [Nov97], but the concept of web-based advertising of generators is the same.

The automatic selection of data structures is an example of *automatic programming* [Bal85]. SETL is a set-oriented language where implementations of sets can be specified manually or determined automatically [Sch81]. SETL offers very few set implementations (bit vector, indexed set, and hashing), and relies on a static analysis of a SETL program using heuristics rather than using cost-based optimizations to decide which set implementation to use. AP5 has comparable capabilities and takes a similar approach [Coh93].

Deductive program synthesis is another way to achieve automatic programming [Bal85, Smi90, Man92, Low94, Kan93]. 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 focussed on generating *algorithms* (e.g., algorithms for solving PDEs [Kan93], algorithms for scheduling [Smi90], algorithms for computing solar incidence angles [Low94], etc.); subroutine libraries are the components from which generated algorithms are built. In contrast, 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). The scale of encapsulation of GenVoca components as well as the way in which domains are modeled (i.e., as realms of plug-compatible components) are different than typical domain theories discussed in the program synthesis literature.

The techniques we used for optimizing type equations are similar to those of rule-based query optimization [Das95, War97]. 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. 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 appears to be more difficult than query optimization.

7 Conclusions

P3 is a GenVoca generator for container data structures. Although its basic technology was developed earlier [Bat93-94], 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). 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 power-

ful — is that P3 produces data structures for a specific application (where all sorts of optimizations can be performed) whereas conventional libraries only offer generic data structures (where these optimizations have not been applied).

The World Wide Web is a powerful and pervasive medium that can significantly alter the landscape of how reuse technology — and generator technology in particular — is delivered to clients. As builders of P3, we obviously want to advertise and disseminate P3 beyond the confines of academic uses. For this reason, we have developed the ContainerStore applet as a visual domain-specific programming language for writing P3 programs.[4] This applet can (soon) be downloaded from our Web site, from which it will communicate (via a server) to the P3 generator and P3 design wizard. The novelty of the ContainerStore applet is its integration of a suite of tools and services that P3 alone cannot provide. The particular services that we discussed in this paper are: english-generated explanations of P3 component compositions, 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 generation of Java code given a P3 specification (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). In future papers, we hope to report on our experiences with the ContainerStore.

Finally, we note that our design wizard is merely a prototype of a much more general technology for automatic component selection and composition. At present, we are using a very simple heuristic (i.e., a greedy algorithm) to search the space of P3 type equations. More work is needed to evaluate whether this heuristic is adequate or if much better designs can be inferred when more powerful search strategies are used. On another front, it is important that design wizards for other (GenVoca-modeled) domains be explored. We believe that analyzing design wizards for different domains may lead to a general model for expressing type equation rewrite rules. Such a model would be very important, as it would 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 penultimate way to minimize software development and maintenance costs through component reuse.

Acknowledgments. We thank Rich Cardone and Yannis Smaragdakis for their comments on earlier drafts of this paper. Our Web site is **`http://www.cs.utexas.edu/users/schwartz/`**.

8 References

[Bal85] R. Balzer, “A Fifteen-Year Perspective on Automatic Programming”, *IEEE Transactions on Software Engineering*, November 1985.

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

[Bat94] D. Batory, J. Thomas, and M. Sirkin, “Reengineering a Complex Application Using a Scalable Data Structure Compiler”, *ACM SIGSOFT 1994*.

[Bat97a] D. Batory and B.J. Geraci, “Validating Component Compositions and Subjectivity in GenVoca Generators”, *IEEE Transactions on Software Engineering*, February 1997, 67-82.

[Bat97b] D. Batory, B. Lofaso, and Y. Smaragdakis, “JTS: A Tool Suite for Building GenVoca Generators”, submitted for publication, 1997.

[Bat97c] D. Batory, “Intelligent Components and Software Generators”, *Software Quality Institute Symposium on Software Reliability*, Austin, Texas, April 1997.

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

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

[Big94] T. Biggerstaff, "The Library Scaling Problem and the Limits of Concrete Component Reuse", *International Conference on Software Reuse*, Rio de Janeiro, November 1-4, 1994, 102-110.

[Bro97] S.V. Browne and J.W. Moore, "Reuse Library Interoperability and the World Wide Web", *Int. Conference on Software Engineering*, May 1997, 684-691.

[Che97] G. Chen, P3 Performance Report, UTCS, University of Texas at Austin, September 1997.

[Coh93] D. Cohen and N. Campbell, "Automating Relational Operations on Data Structures", *IEEE Software*, May 1993.

[Das95] D. Das and D. Batory, "Prairie: A Rule Specification Framework for Query Optimizers". *International Conference on Data Engineering*, Taipei, March 1995, 201-210.

[Due97] A. Van Duersen and P. Klint, "Little Languages: Little Maintenance?", *Proc. First ACM SIGPLAN Workshop on Domain-Specific Languages*, Paris 1997.

[Eas73] C.M. Eastman, "Automated Space Planning", *Artificial Intelligence 4(1973)*, 41-64.

[Gog86] J.A. Goguen, "Reusing and Interconnecting Software Components", *IEEE Computer*, February 1986.

[Jen97] 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.

[Kan93] E. Kant, et al., "Synthesis of Mathematical Modeling Software", *IEEE Software*, May 1993, 30-41.

[Kie96] R. Kieburtz, et al., "A Software Engineering Experiment in Software Component Generation", *International Conference on Software Engineering*, 1996.

[Lad97] R. Laddaga, *Self-Adaptive Software Workshop*, Kestrel Institute, July 1997.

[Low94] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood, "AMPHION: Automatic Programming for Scientific Subroutine Libraries". *Intl. Symp. on Methodologies for Intelligent Systems*, Charlotte, North Carolina, Oct. 16-19, 1994, 326-335.

[Man92] Z. Manna and R. Waldinger, "Fundamentals of Deductive Program Synthesis", *IEEE Trans. Software Engineering*, August 1992, 674-704.

[Mit75] M.F. Mitoma and K.B. Irani, "Automatic Database Schema Design and Optimization", *1975 Very Large Databases Conference*, 286-321.

[Nov95] G. Novak, "Automatic Programmer Server", **`http://www.cs.utexas.edu/novak`**, 1995.

[Nov97] G. Novak, "Software Reuse by Specialization of Generic Procedures through Views", *IEEE Trans. Software Engineering*, July 1997, 401-417.

[Sch81] 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*, April 1981, 126-143.

[Sma97] Y. Smaragdakis and D. Batory, "DiSTiL: a Transformation Library for Data Structures", *USENIX Conf. on Domain-Specific Languages*, 1997.

[Smi90] D.R. Smith, "KIDS: A Semiautomatic Program Development System", *IEEE Trans. Software Engineering*, September 1990, 1024-1043.

[Sre97] S. Sreerama, D. Fleming, M. Sitaraman, "Graceful Object-Based Performance Evolution", *Software Practice and Experience*, January 1997.

[Ste94] A.A. Stepanov and M. Lee, "The Standard Template Library", HP Laboratories, TR HPL-94-34, 1994.

[War97] L. Warshaw, D. Miranker, and T. Wang, "A General Purpose Rule Language as the Basis of a Query Optimizer", UTCS TR97-19, University of Texas at Austin, July 1997.

[Pou95] J.S. Poulin and K.J. Werkman, "Melding Structured Abstracts and the World Wide Web for Retrieval of Reusable Components", *Symposium on Software Reusability* (1995), 160-168.

[X3M97] X3M Solutions, "CAL: The Container and Algorithm Library for the Java Platform", URL: **`http://www.x3m.com/products/cal/`**, 1997.