Composition Validation and Subjectivity in GenVoca Generators

Don Batory, *Member*, *IEEE*, and Bart J. Geraci, *Member*, *IEEE*

Abstract—GenVoca generators synthesize software systems by composing components from reuse libraries. GenVoca components are designed to export and import standardized interfaces, and thus be plug-compatible, interchangeable, and interoperable with other components. In this paper, we examine two different but important issues in software system synthesis. First, not all syntactically correct compositions of components are semantically correct. We present simple, efficient, and domain-independent algorithms for validating compositions of GenVoca components. Second, components that export and import immutable interfaces are too restrictive for software system synthesis. We show that the interfaces and bodies of GenVoca components are subjective, i.e., they mutate and enlarge upon instantiation. This mutability enables software systems with customized interfaces to be composed from components with "standardized" interfaces.

Index Terms—GenVoca, software generators, subjectivity, compositon validation, design rule checking.

1 Introduction

SOFTWARE system generators automate the development of software for large families of applications. Generators automatically transform compact, high-level specifications of target systems into actual source code, and rely on libraries of parameterized, plug-compatible, and reusable components for code synthesis.

Generators [10], [1], [8], [24], [37], [40] are among many approaches that are being explored to construct customized software systems quickly and inexpensively from reuse libraries. CORBA and its variants simplify the task of building distributed applications from components [56]; CORBA can simplify the manual integration of independently-designed and standalone modules in a heterogeneous environment. In contrast, generators are closer to toolkits [25], object-oriented frameworks [34], and other reuse-driven approaches (e.g., [58], [52]), because they focus on software domains whose components are not standalone, that are designed to be plug-compatible and interoperable with other components, and that are written in a single language. The particular class of generators that we consider in this paper, called *GenVoca generators* [1], is distinguished from the above approaches in that its components are parameterized program transformations that encapsulate consistent data and operation refinements. These components also encapsulate logic to automate domain-specific decisions about when to use a particular algorithm and when to apply a domain-specific optimization. For many domains, such decisions are essential for generating efficient code.

This paper addresses two rather different but central problems of GenVoca generators:

1) How does one validate a composition of components?
2) How is interface variability of generated software achieved through component composition?

Assuming that components are implemented correctly, a fundamental problem for all component-based software development technologies is: Does a composition of components meet the behavioral (or functional) specifications of the target system? For the case of GenVoca generators, this is the problem of *design rule checking*, i.e., the detection of illegal combinations of components. To be viable tools of future software development environments, it is critical that generators validate component compositions automatically (and suggest repairs when errors are detected), rather than burdening users with the impossible task of debugging generated code.

In the first part of this paper, we present domain-independent algorithms for design rule checking in GenVoca generators, and the domain-specific variants that we used in the P2 and Genesis projects. Our work is related to Perry's Inscape environment, which (among other topics) dealt with consistency checking in software composition models [47], [48]. We adapt and generalize the component consistency checking approach of Inscape to exploit the semantics of layers in the construction of hierarchical software systems. We explain how GenVoca models of software domains are grammars, where sentences correspond to component compositions. By encoding component properties as inherited and synthesized attributes, we find that attribute grammars provide a natural formulation of the legal sentences (component compositions, software systems) of a domain. We illustrate our results by explaining how the P2 data structure generator validates component compositions.

Another fundamental problem in software component technologies is: how can the variability of interfaces of systems within a domain be explained and synthesized? All systems of a software domain do not export the same interface; there will always be variations. Ossher and Harrison

- *D. Batory and B.J. Geraci are with the Department of Computer Sciences, University of Texas, Austin, Tx 78712. E-mail: {dsb, geraci}@cs.utexas.edu.*

Manuscript received Feb. 12, 1997.
Recommended for acceptance by S.H. Zweben and M. Sitaraman.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number S97021.

call this variability subjectivity [30], [29], [44], [45]: No single interface can adequately describe any object that is common to a family of applications. Such objects must be described by a family of interfaces; the particular interface that is appropriate for an object for a given application is *subjective* (i.e., application-dependent).

In the second part of this paper, we explore the relationship of subjectivity to GenVoca. (We believe that subjectivity impacts *all* generators, but here we focus exclusively on its impact on GenVoca.) We show that typical component interfaces (i.e., ones that are cast-in-concrete and that do not change upon instantiation) are far too rigid to be practical; GenVoca components have interfaces and bodies that enlarge automatically upon instantiation and hence are subjective (i.e., system-dependent). We review techniques that have been used to achieve subjective interfaces in four independently-conceived generators and present a model that unifies them.

2 THE GENVOCA MODEL OF SOFTWARE SYSTEM GENERATION

GenVoca is a domain-independent model for defining scalable families of hierarchical systems from components. Its basic premise is that standardizing both the fundamental abstractions of mature software domains *and* their implementations, one can define plug-compatible and interchangeable software "building blocks." Although the number of fundamental abstractions in a domain is rather small, there is a huge number of potential implementations. GenVoca also advocates a layered decomposition of implementations, where each layer or component encapsulates a primitive domain feature. The advantage of GenVoca is *scalability* [3], [9]: Component libraries are relatively small and grow at the rate new components are entered, whereas the number of possible *combinations* of components (i.e., distinct software systems in the domain that can be defined) grows geometrically. Generators that use GenVoca organizations have been built for the domains of avionics, data structures, databases, file systems, and network protocols [13], [3], [30], [32].

Components and Realms. A hierarchical software system is defined by a series of progressively more abstract virtual machines [16]. (A *virtual machine* is a set of classes, their objects, and methods that work cooperatively to implement some functionality. Clients of a virtual machine do not know how this functionality is implemented.) A *component* or *layer* is an implementation of a virtual machine. The set of all components that implement the same virtual machine is a *realm*; effectively, a realm is a library of plug-compatible and interchangeable components. In Fig. 1a, realms `S` and `T` have three components, whereas realm `W` has four.

```
S = { a, b, c }
T = { d[S], e[S], f[S] }
W = { n[W], m[W], p, q[T, S] }
```
(a)
```
S := a | b | c ;
T := d S | e S | f S ;
W := n W | m W| p | q T S ;
```
(b)

Fig. 1. Realms, components, and grammars.

Parameters and Transformations. A component has a (realm) parameter for every realm interface that it imports. All components of realm `T`, for example, have a single parameter of realm `S`.[1] This means that every component of `T` exports the virtual machine interface of `T` (because the component belongs to realm `T`) and imports the virtual machine interface of `S` (because it has a parameter of realm `S`). Thus, each `T` component encapsulates a mapping or *transformation* between the virtual machines `T` and `S`. Such transformations often involve domain-specific optimizations and the automated selection of appropriate algorithms.

Systems and Type Equations. A software *system* is modeled by a named composition of components called a type *equation*. Consider the following two equations:

```
System_1 = d[b];
System_2 = f[a];
```

`System_1` is a composition of component `d` with `b`; `System_2` composes `f` with `a`. Note that both systems are equations of type `T` (because the outermost components of both systems are of type `T`). This means that both implement the same virtual machine and hence, `System_1` and `System_2` are interchangeable implementations of the interface of `T` (with respect to functionality, not performance). Note that composing components is equivalent to stacking layers in hierarchical systems. We will use the terms component and layer interchangeably in this paper.

Grammars, Families of Systems, and Scalability. Realms and their components define a grammar whose sentences are software systems. Fig. 1a enumerated realms `S`, `T`, and `W`; the corresponding grammar is shown in Fig. 1b. Just as the set of all sentences defines a language, the set of all component compositions defines a *family of systems*. Adding a new component to a realm is equivalent to adding a new rule to a grammar; the family of systems enlarges automatically. Because large families of systems can be built using few components, GenVoca is a *scalable* model of software construction.

Symmetry. Just as recursion is fundamental to grammars, recursion in the form of *symmetric* components is fundamental to GenVoca. More specifically, a component is symmetric if it exports the same interface that it imports (i.e., a symmetric component of realm `W` has at least one parameter of type `W`). Symmetric components have the unusual property that they can be composed in almost arbitrary ways. In realm `W` of Fig. 1, components `n` and `m` are symmetric whereas `p` and `q` are not. This means that compositions `n[m[p]]`, `m[n[p]]`, `n[n[p]]`, and `m[m[p]]` are possible, the latter two showing that a component can be composed with itself. In general, the order in which components are composed can significantly affect the performance of the resulting system [4], [53].

Design Rules and Domain Models. In principle, any component of realm `S` can instantiate the parameter of any component of realm `T`. Although the resulting equations would be *type correct*, the equation may not be semantically correct. That is, there are often domain-specific constraints *in addition to* implementing a particular virtual machine that

1. Components may have other parameters in addition to realm parameters. In this paper, we focus only on realm parameters.

instantiating components must satisfy. These additional constraints are called *design rules*. *Design rule checking (DRC)* is the process of applying design rules to validate type equations. A *domain model* for a GenVoca generator consists of realms of components and design rules that govern component composition.

3 PART I: DESIGN RULE CHECKING IN GENVOCA GENERATORS

What exactly is the form that design rules should take? How complicated are typical design rules? Are there different kinds of rules? Can design rule checking be done automatically, or will human guidance be needed? To answer these questions, we briefly review the domain model of the P2 generator and illustrate some of its design rules. We then develop a model of DRC and outline algorithms based on attribute grammars that rely on shallow consistency checking.

3.1 P2 Domain Model

P2 is a GenVoca generator for container data structures [3], [5]. The domain model of P2 relies on two realms: `ds` and `mem` (see Fig. 2). `ds` components export a standardized container-cursor interface. Among the components of `ds` are those that implement common data structures (e.g., binary trees, doubly linked ordered and unordered lists) and storage options (e.g., free lists of deleted elements, sequential and random storage). `mem` components export standardized memory allocation and deallocation operations. Among its members are components that manage space in transient and persistent memory.

In Fig. 2, we list only the realm parameters of the `ds` and `mem` components. For example, the `bintree` component links elements of a container onto a binary tree. The key of the binary tree is a nonrealm parameter of `bintree` that is not shown. Similarly, the `persistent` component has an additional parameter, i.e., the name of the persistent file in which elements are to be stored, and so too does `array`, i.e., the size of the array to allocate. Nonrealm parameters are definitely present in any implementation of components (including P2's). Constraints on nonrealm parameters can be tested locally within a component; this is not a difficult or interesting problem. Our focus on design rule checking is on realm parameters and constraints on realm parameters that cannot be satisfied locally.

Type Equations. A P2 type equation is a composition of one or more `ds` and `mem` components that defines how elements of a container are to be stored. A P2 type equation `a[b[c]]` is evaluated *outermost* component first—i.e., component/transformation `a` is applied first, then `b`, and lastly `c`. This is different than a typical "functional" evaluation where the innermost term is evaluated first and the outermost last. Intuitively, the reason is that a P2 program is an abstract specification of an application that manipulates data structures. A type equation maps an abstract P2 program to a concrete C program by a sequence of transformations that progressively reveal container data structure implementation details. The first transformation to apply (by convention) is the outermost component; the last transformation is the innermost. The following examples give a flavor of the kinds of data structures that can be expressed.

EXAMPLE 1. Consider equation `ex1`:

```
ex1 = conceptual[
        bintree[ heap[ transient ] ] ];
```

`conceptual` is a common first component of all P2 type equations; it is responsible for generating code fragments that are shared by many data structures. `bintree` links elements of a container onto a binary tree, `heap` assigns heap addresses to elements, and `transient` allocates element storage in transient memory. Interpreting the components of equation `ex1` from outermost to innermost means that `ex1` defines a container data structure that links elements onto a binary tree, whose nodes are stored in a heap in transient memory.

EXAMPLE 2. Type equation `ex2` defines the same data structure as `ex1`, except that the container and its elements are stored in persistent memory. This is accomplished by swapping the `transient` memory component with the `persistent` component:

```
ex2 = conceptual[
       bintree[ heap[ persistent ] ] ];
```

EXAMPLE 3. Type equation `ex3` defines a container data structure that stores elements on a binary tree, whose nodes are linked together on a key-ordered doubly-linked list and are stored in a heap in transient memory:

```
ex3 = conceptual[ bintree[
       odlist[ heap[ transient ] ] ] ];
```

```
ds = { bintree[ ds ],    // binary tree
       dlist[ ds ],      // unordered doubly linked list
       odlist[ ds ],     // key-ordered list
       avail[ ds ],      // free list of deleted elements
       index[ ds, ds ],  // key indexing
       heap[ mem ],      // heap storage
       array[ mem ],     // sequential (array) storage
       qualify[ ds ],    // qualify retrieval of elements
       inbetween[ ds ],  // deletion actions
       top2ds[ ds ],     // first layer of a ds expression
       conceptual[ ds ], // composition of top2ds[inbetween[qualify[…]]]
       … }

mem = { transient,       // transient memory
        persistent,      // persistent memory
        … }
```

Fig. 2. The `ds` and `mem` realms.

EXAMPLE 4. A container that simultaneously links elements on two distinct key-ordered doubly-linked lists where deleted nodes are recycled on an `avail` list and nodes are stored sequentially in an `array` in `transient` memory is expressed by equation `ex4`:

```
ex4 = conceptual[ odlist[ odlist[
      avail[ array[ transient ] ] ] ] ];
```

From these few examples, it is not difficult to see that a small number of `ds` and `mem` components can be composed in many ways to yield large families of distinct and complex data structures. This is typical of the scalability of GenVoca domain models.

Design Rules. Currently there are over 50 components in P2, most of which are symmetric. As our examples suggest, container data structures are defined by type equations that reference up to 20 components. Although interpreting equations (in the outer-to-inner manner) is straightforward, knowing whether an equation defines a correct composition is not always evident. Validation is complicated by the fact that many components have nonobvious rules for their use.

As an example, consider the `inbetween` component, which is one the components of the `conceptual` component "macro." `inbetween` encapsulates algorithms that are shared by many data structure components (e.g., `bintree` and `odlist`). The `inbetween` algorithms deal with the positioning of a cursor immediately after an element has been deleted—does the cursor point to a "hole" in the container where the deleted element resided or should it be positioned on the next element in the container? Instead of replicating these algorithms in every data structure component (and then dealing with the maintenance/consistency problems that would ensue), the algorithms are written once as the `inbetween` component. A consequence of this factoring is that a precondition for using a data structure component is the previous appearance of `inbetween` in a type equation. More specifically, the valid use of `inbetween` requires that a *single* copy of `inbetween` be present in a type equation that uses at least one data structure component (`odlist`, `bintree`, etc.) and it should be outside of *all* such components in the equation. The right equation, below, shows a correct usage—i.e., `inbetween` is outside of all data structure components. The `wrong` equation, below, shows an incorrect usage: a data structure component `odlist` appears prior to `inbetween`.

```
right = …inbetween[…[odlist[odlist[…]]]]…;
wrong = …odlist[…[inbetween[odlist[…]]]]…;
```

Rules such as this should not be borne by programmers; they are *much* too easy to forget and to be misapplied. A design rule checker that tests such rules automatically and reports errors when they occur removes a great burden from P2 users. We first present a general model of design rule checking in Section 3.2 and then show how we adapted the model to P2 and Genesis generators in Section 3.3 and Section 3.4.

3.2 A Model of Design Rule Checking

Perry's Inscape is an environment for managing the evolution of software systems [46], [47], [48]. Among the features it supports is consistency checking, a simplified form of verification. Components (i.e., operations) have preconditions for their use and postconditions (that describe what is known to be true as a result of an operations's execution). A novel aspect of Inscape is that components additionally have *obligations* which are conditions that must be satisfied by any system that uses a component. Obligation predicates require "action-at-a-distance": Although they *might* be satisfied locally by adjacent components, generally they depend on global properties of the system (i.e., on properties of nonadjacent components). Obligations are propagated to their enclosing modules where eventually they must be satisfied by some postconditions. Another aspect of Inscape is that full-fledged verification is not attempted. Instead, primitive predicates are declared and informally defined, typically with their names hinting at their semantics. Preconditions, postconditions, obligations are expressed in terms of these predicates, thus enabling a practical but powerful form of "shallow" consistency checking to be achieved using pattern matching and simple deductions.

The Inscape approach can be adapted to design rule checking by exploiting the semantics of layers. First, design rule checking examines the static properties of software system (type equation) development; it does not model states of system execution. Fig. 3 illustrates the distinction. Suppose `s[Q]` is a system that is parameterized by realm `Q`. Suppose further that `k` is a component of `Q`. Composing `s` with `k` maps system `s` to system `s' = s[k]`. To model states of system (type equation) development, every system is described by a set of attributes whose values define its states or properties. Thus, we might define an attribute `State` whose value is `no-loops` in system `s` (meaning that `s` has no loops), and after the instantiation, `State` has the value `has-loops` (meaning that `s'` has loops). *Design rule checking deals with the testing and assignment of static properties of system designs; it assumes that all transformations (components) are semantically correct.*

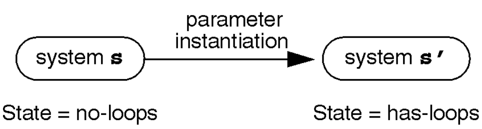

Fig. 3. Modeling states of program development.

Second, it is common for GenVoca components to have preconditions and obligations that are not satisfied locally, i.e., by components that are adjacent to it in a type equation. Preconditions and obligations of a component `k` are satisfied "at-a-distance," that is, by components that either lie (far) beneath `k` or (far) above `k` in a type equation.[2] Moreover, the properties exported by `k` to "higher" layers are generally not the same properties that are exported to "lower" layers. For this reason, we found it necessary to distinguish two kinds of preconditions and postconditions.

Postconditions are properties of `k` that are to be exported to components *beneath* `k` in a type equation. *Preconditions* define the properties that must hold for `k` to work properly; they test the cumulative postconditions of components that lie *above* `k` in a type equation.

2. We use the terms "higher" and "lower" to refer to positions of components within a type equation. The outermost component of an equation is the "highest" component, and the innermost components are the "lowest."

EXAMPLE. Suppose component **k** has a precondition that attribute **A** must have the value **v** (see Fig. 4a). For **k** to be used correctly, there must be some component, say **u**, that sits above **k** whose postcondition sets **A = v**. Note that **u** need not be immediately above **k**; **u** might reside far above **k**.

As a concrete example, recall the type equations **right** and **wrong** of Section 3.1 that involve compositions of the **inbetween** and **odlist** components. Let attribute **inbetween_present** represent the property that the **inbetween** component has been used in a type equation. The **inbetween** component asserts this property as the postcondition **inbetween_present = assert**. A precondition for using **odlist** is that there is an **inbetween** layer that lies above it in a type equation. This precondition is **inbetween_present = assert**. Thus, an equation that has a data structure component (e.g., **odlist**) but no **inbetween** component is incorrect.

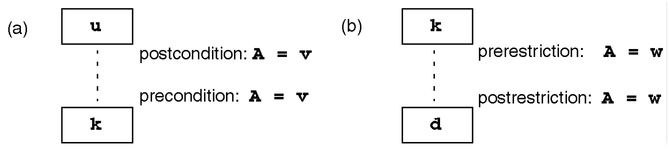

Fig. 4. Different kinds of design rules.

Postrestrictions are properties of **k** that are to be exported to components *above* **k** in a type equation. *Prerestrictions* (which correspond to Inscape obligations) are preconditions for instantiating component parameters; they test the cumulative postrestrictions of components that lie *beneath* **k** in a type equation.

EXAMPLE. Suppose component **k** has a single parameter with the prerestriction that attribute **A** must have the value **w** (see Fig. 4b). For the parameter to be correctly instantiated, there must be some component, say **d**, that lies below **k** whose postrestriction sets **A = w**. Analogously, **d** need not be immediately beneath **k**; **d** might reside far below **k**.

As a concrete example, again recall the **inbetween** and **odlist** components. Let attribute **data_struct _present** denote the property that a data structure component (like **bintree** and **odlist**) is present in a type equation. **odlist** would assert this property to higher level components as the postrestriction **data_struct_present = assert**. For **inbetween** to be used correctly in a type equation, there must be at least one data structure component beneath **inbetween**; this is expressed by the prerestriction **data_struct_present = assert**. Thus, a type equation that has an **inbetween** component but no data structure component (e.g., **odlist**) is incorrect.

Given GenVoca design rules (i.e., preconditions, postconditions, prerestrictions, and postrestrictions) of every component of a type equation, design rule checking involves:

- a top-down propagation of postconditions and the testing of component preconditions, and
- a bottom-up propagation of postrestrictions and the testing of parameter prerestrictions.

In the following sections, we present general algorithms for top-down and bottom-up design rule checking. We initially place no restrictions on the complexity of DRC predicates. Later in Section 3.3, however, we show that predicates for domain-customized instances of our algorithms are very simple and are consistent with the shallow consistency checking approach taken in Inscape [46], [47], [48].

3.2.1 Top-Down Design Rule Checking

Consider component **k[x]** which has a single parameter **x**. **k** has both a precondition (**precondition-k**) and a postcondition (**postcondition-kx**). Let **top** denote the set of attribute values that are known to hold at the point immediately above **k** in a type equation. Component **k** is correctly used if **top** implies **k**'s preconditions (i.e., **top** $\Rightarrow$ **precondition-k**). The set of attribute values that hold immediately beneath k in the type equation is computed by applying the postconditions of **k** to the current conditions (i.e., **top-x = postcondition-kx** $\oplus$ **top**). The operator $\oplus$ is the *postcondition propagation operator*. When type equations correspond to a linear stack of components, the testing of preconditions and the propagation of postconditions is straightforward: Only two operators $\oplus$ and $\Rightarrow$ are needed.

In general, type equations are trees of components. Branching arises when components have multiple parameters, e.g., **d[x, y]**. Each parameter of a component has its own postcondition that defines the set of attribute values that hold for that parameter; these are the values that are propagated to any system instantiating that parameter. In the case of component **d[x, y]**, parameter **x** would have **postcondition-dx** as its postcondition and parameter **y** would have **postcondition-dy**. Let **top** be the set of conditions that hold prior to component **d** in a type equation, **top-x** be the set of conditions that hold for parameter **x** after **d** has been applied, and **top-y** be the set of conditions that hold for parameter **y**. **top-x** is computed by applying **x**'s postcondition to **top** (i.e., **top-x = postcondition-dx** $\oplus$ **top**) and **top-y** is computed similarly (**top-y = postcondition-dy** $\oplus$ **top**). Given the operators $\oplus$ and $\Rightarrow$, there is a straightforward, recursive algorithm for the top-down propagation of postconditions and the testing of component preconditions [6].

3.2.2 Bottom-Up Design Rule Checking

Every parameter of a component has prerestrictions for instantiation, and every component has postrestrictions that are exported to higher layers in a type equation. Fig. 5 depicts a typical situation: components **q**, **r**, **s**, **t**, and **w** are composed hierarchically, and **q** has a single parameter. In general, the prerestrictions for **q** are not satisfied by the component **r** that instantiates its parameter, but rather by components deep within the system rooted at **r**. That is, the prerestrictions of **q** may be satisfied by **r** or **s** or **t** or **w**, or any combination thereof.

This leads to the interpretation of instantiation that *systems* instantiate parameters, not components. Every system exports a realm interface plus postrestrictions that higher layers can reference. A component parameter is correctly instantiated if the postrestrictions of the instantiating system imply that parameter's prerestrictions.

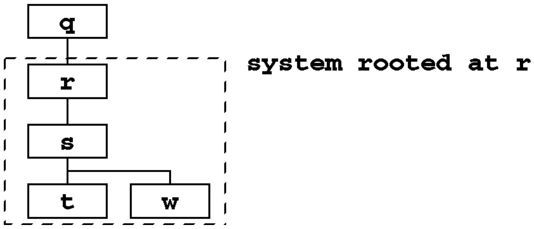

Fig. 5. System instantiation of parameters.

Consider component **u[x]**. **u** has both a prerestriction (**prerestriction-ux**) and a postrestriction (**postrestriction-u**). Let **bottom** denote the set of attribute values that are exported by a system that instantiates parameter **x**. **x** is instantiated correctly if **bottom** implies its prerestrictions (i.e., **bottom** ⇒ **prerestriction-ux**). The set of attribute values that are exported by the system rooted at **u** is computed by applying the postrestrictions of **u** to the attribute values of the system that it imported (i.e., **bottom' = postrestriction-u ⊕ bottom**). Note that the same operators ⇒ and ⊕ used in top-down design rule checking are used in bottom-up design rule checking. Just as in the case of top-down design rule checking, there is a simple, recursive algorithm for the bottom-up propagation of postrestrictions and the testing of parameter prerestrictions [6].

3.2.3 Attribute Grammars

McAllester [38] observed that attribute grammars unify realms, components, attributes, top-down and bottom-up design rule checking. From previous sections, we know that realms of components define a grammar. Attributes model static properties of system (type equation) development. Postconditions assign values to inherited attributes (i.e., attributes whose values are determined by component ancestors) and postrestrictions assign values to synthesized attributes (i.e., attributes whose values are determined by component descendants). The practical benefit of this connection with attribute grammars, besides the fact that design rule checking reduces to a well-studied problem, is that common tools, such as **bison** and **yacc**, are well-suited for writing design rule checkers, as we will see in Section 3.4.

3.3 Targeting DRC Algorithms to Specific Domains

The design rule checking algorithms of Section 3.2 are domain-independent. To specialize them to a particular domain, we need definitions and representations for attributes, predicates, and the operators ⊕ and ⇒. In the following, we explain the representations that we implemented for P2; virtually the same representations were used in Genesis.

3.3.1 Attributes

An attribute models a property that exposes a composition constraint. Although the properties in which we are interested undoubtedly have complex formal definitions, we have found (like Perry [46], [47], [48]) that in practice they can be defined informally as attributes that assume restricted values. The values we use (**any**, **assert**, **negate**, and **inherit**) are defined in Table 1.

TABLE 1
ATTRIBUTE VALUES USED IN P2 AND GENESIS

Attribute Value	Interpretation
any	nothing is known
assert	property is asserted
negate	property is negated
inherit	property value is inherited from existing conditions

Example P2 attributes are: **logical_deletion** and **retrieval**. **logical_deletion** represents the property that a component implements logical deletions. That is, instead of physically deleting an element from a container, the component marks the element deleted but does not immediately reclaim its space. The **retrieval** attribute represents the property that a component interlinks all elements of a container to facilitate searching. Components that implement data structures (e.g., **bintree**, **dlist**, etc.) have the **retrieval** property. The assignment of **assert** or **negate** to these attributes as a postcondition or postrestriction depends on whether a component satisfies the property. **inherit** is used when the value of an attribute is unchanged by a component.

3.3.2 Predicates

Preconditions and prerestrictions in P2 and Genesis request specific attribute values (e.g., **any**, **assert**, **negate**), but not how the attribute value was determined (e.g., **inherit**). Table 2 lists the four different primitive predicates that can be defined over a *single* attribute. P2 predicates are conjunctions and disjunctions of these primitive predicates. Conjunctive predicates are implemented as a vector of primitive predicates that are indexed by attribute. Thus, predicate $P_1 \wedge P_2 \wedge \ldots \wedge P_n$ is encoded as the vector $[P_1, P_2, \ldots, P_n]$ where P_i is the primitive predicate for attribute *i*.

TABLE 2
PRIMITIVE PREDICATES USED IN P2 AND GENESIS

Predicate	Interpretation
P-any	true (no constraints)
P-assert	attribute has **assert** value
P-negate	attribute has **negate** value
P-false	false (unsatisfiable)

3.3.3 Postcondition Propagation Operator ⊕

Component postconditions and postrestrictions selectively declare new attribute values (e.g., **assert** or **negate**) or propagate existing (**inherited**) values. Table 3 defines the condition propagation operator + for a single attribute. Given a postcondition / postrestriction value vector $V = [V_1, V_2, \ldots V_n]$ and the vector of existing conditions $E = [E_1, E_2, \ldots, E_n]$, the ⊕ operator is vector addition using the + operator of Table 3:

$$V \oplus E = [V_1 + E_1, V_2 + E_2, \ldots, V_n + E_n]$$

TABLE 3
THE PROPAGATION OPERATOR + FOR A SINGLE ATTRIBUTE

v + e		E value		
		any	assert	negate
V value	assert	assert	assert	assert
	negate	negate	negate	negate
	inherit	any	assert	negate

3.3.4 Implication Operator ⇒

The implication operator → for a *single* attribute is defined by a truth-table (Table 4). Given a vector of existing conditions E = [E_1, E_2, …, E_n] and a precondition / prerestriction vector P = [P_1, P_2, …, P_n] of a conjunctive predicate, the implication operator ⇒ has a simple definition: all primitive predicates must be true for the compound predicate to be true. (A simple generalization handles disjunctions.)

$$E \Rightarrow P = (E_1 \rightarrow P_1) \wedge (E_2 \rightarrow P_2) \wedge \ldots \wedge (E_n \rightarrow P_n)$$

TABLE 4
THE IMPLICATION OPERATOR → FOR A SINGLE ATTRIBUTE

E → P		P value			
		P-any	P-assert	P-negate	P-false
E value	any	true	false	false	false
	assert	true	true	false	false
	negate	true	false	true	false

3.4 Implementation Notes

The implementation of our DRC algorithms and the P2 / Genesis specializations of the ⊕ and ⇒ operators was straightforward: the source files consist of 1,500 lines of `lex` and `yacc`. We wrote a general utility, called `dreck`, that would allow designers to declare realms, components, and their design rules based on the representations we noted previously for attributes, predicates, and DRC operators [6]. Fig. 6 shows a `dreck` declaration of the `array` component and its design rules. A component's name, realm membership, and realm parameters are declared on the first line. Subsequent lines define design rules. A precondition for `array`'s usage is that a layer above `array` needs to support logical deletion. This precondition is expressed by asserting the `logical_deletion` property. Other design rules assert to layers above and below that `array` is a retrieval layer. Such declarations are expressed by asserting the `retrieval` property as a postcondition and postrestriction.

```
                         name of component
                         realm of component
                         component parameters
                         design rules
array : ds [ mem ] {
   # logical deletion layer required
   # above array

   precondition assert logical_deletion

   # assert that array is a retrieval
   # layer to all descendants and
   # ancestors

   postcondition   assert retrieval
   postrestriction assert retrieval
}
```

Fig. 6. Specification of design rules.

Algorithm Efficiency. Let n denote the number of components in a type equation and let m denote the number of attributes. A straightforward implementation of the DRC algorithms is as a tree traversal, where each node is visited twice (once on the way down from the root, and once on the way up from visiting leaves). At each visit, m attribute values are propagated. Thus, the complexity of our algorithm is $O(mn)$.

To give readers upper estimates of n and m, the most complicated type equations that we have encountered in Genesis and P2 have approximately 30 components (i.e., $n \le 30$). Genesis maintains the greatest number of attributes ($m = 14$), whereas P2 has fewer ($m = 8$), even though both generators have libraries of 50 components. Although it is not difficult to envision greater values for `m` and `n`, substantially greater values (e.g., $m, n > 100$) seem unlikely.

Extensibility. Adding new components to a domain model is not difficult. The component designer must determine whether existing attributes are sufficient to capture illegal compositions (in which case component addition is trivial) or whether new attributes are needed. In practice, adding more attributes has not been problematic because the number of components in generator libraries is modest (and because of scalability, we would expect the number to remain small). For example, ADAGE has the largest library (about 400 components) which avionics experts have no difficulty managing.

3.5 Explanation-Based Error Reporting

Detecting composition errors is only part of the problem of debugging type equations; repairing equations are also important. *Precondition ceilings* is a technique used in Inscape that we found particularly effective. Suppose component `Y`'s precondition `A = v` failed. This means that some component above `Y`, say `X`, set `A ≠ v` as a postcondition. To repair this error, there needs to be another component, `Z`, that must be inserted below `X` and above `Y` whose postcondition is `A = v`. Techniques such as this (including obligation / prerestriction ceilings) form the basis of a powerful explanation-based error reporting scheme. The following example illustrates the idea.

EXAMPLE. Suppose we would like a P2 container that stores elements in a binary tree, whose nodes are stored sequentially in an array in transient memory. A first attempt at its type equation might be:

```
first_try = top2ds[bintree[array[transient]]];
```

Our DRC algorithms report the following:

```
Precondition errors:
   an inbetween layer is expected between
      top2ds and bintree
   a logical deletion layer is expected
      between top2ds and array
Prerestriction error:
   parameter 1 of top2ds expects a
      subsystem with a qualification layer
```

The first error reminds us (from Section 3.1) that we forgot that a `bintree` layer requires the `inbetween` layer to be above it. Not only that, the error message states exactly how to repair the equation; there is only one location where `inbetween` can go (i.e., in between `top2ds` and `bintree`). The second error reminds us that `array` requires a logical deletion layer above it. Further, this layer must be below `top2ds`. The third error tells us that a qualification layer is required below `top2ds`. Users with minimal experience with P2 are able to repair all of these errors easily. But suppose repairs lead to the following equation:

```
second_try = top2ds[inbetween[bintree[
             qualify[delflag[array[
             transient]]]]]];
```

where `qualify` is a qualification layer and `delflag` is a logical deletion layer. The DRC response to this equation is:

```
Precondition error:
  a retrieval layer (bintree) is not
    expected above qualify
```

This error tells us that all retrieval layers must lie beneath `qualify`; the fix is to transpose `bintree` and `qualify`, which results in a correct equation:

```
correct = top2ds[inbetween[qualify[bintree[
          delflag[array[transient]]]]]];
```

which readers may recognize as:

```
correct = conceptual[bintree[delflag[array[
          transient]]]];
```

In general, DRC error messages direct users to modify an incorrect equation to the nearest set of correct type equations in the space of all equations. We have found this advice works well. With minimal experience, P2 users typically come very close to their desired equation on the first attempt; DRC messages enable them to correct errors quickly.

3.6 Related Work and Insights

Related Work. DRACO used a form of shallow consistency checking (called assertions and conditions) in composing layers of transformations [40]. DaTE, the design rule checker for Genesis [2] supported only component preconditions. The limitations of DaTE led to the work presented in this paper.

McAllester developed a functional programming language, VAG, based on variational attribute grammars, to address the design rule checking issues for the ADAGE generator [38]. Preconditions and prerestrictions were treated uniformly as constraints. The constraints associated with a component were expressed as a VAG program. When an avionics system was composed from components, the set of constraints that had to be satisfied was defined by the composition of corresponding VAG programs. The VAG interpreter had limited reasoning abilities to infer values of unbound VAG program parameters.

Parameterized programming is intimately associated with the verification of component compositions. Goguen's work on OBJ and library interconnection languages, such as LIL and LILEANNA [21], [22], [55], are basic. The RESOLVE project explores the design of reusable and parameterized components, component certifiability, and the certifiability of component compositions [52], [53]. Although there are many similarities among these works and ours, there is a basic difference: there is no "action-at-a-distance" in the other work. Vertical compositions of OBJ, LILEANNA, and RESOLVE components are verified locally; components constrain the behavior of immediately adjacent components, and not components that reside far above or below them in a hierarchy.

Our work is also an example of the types of consistency checking problems encountered in software architectures [49], [19], [20], [39]. To our knowledge, other than Inscape, validating compositions of components in the context of architectures has only begun to be addressed.

Insights. Our work on DRC was actually developed independently of DRACO and Inscape. That our results are so similar is encouraging: we suspect that "shallow" consistency checking is a general technique for automatic software system generation.

An important distinction between Inscape and our work is the scale of componentry. An Inscape component is a function; a GenVoca component is a subsystem (i.e., a suite of interrelated classes). Perry noted that there can be many primitive predicates when there are thousands or tens of thousands of functions in a system. In contrast, type equations rarely reference more than 50 components, and the number of primitive predicates that we have encountered in modeling different and multiple domains is modest. So, it would seem that scaling the size of a component *reduces* the number of primitive predicates (attributes) that need to be maintained. This seems counterintuitive.

Our best explanation for this centers on two observations. First, we believe that modeling static properties of software system development (instead of states of execution) reduces the number of properties to examine. Second, we believe that GenVoca offers a powerful methodology for the design of reusable components. Object-oriented design methodologies, for example, are powerful because of their ability to manage and control software complexity [11]. It is not difficult to recognize that standardizing domain abstractions and their programming interfaces (i.e., the core of GenVoca) is also a powerful way of managing and controlling the complexity of software in a *family of systems*. We believe that standardization makes some problems tractable that would otherwise be very difficult. Composibility of software components is one example (c.f., [20]) and DRC is another (c.f. [35]).

4 Part II: Subjectivity in GenVoca Generators

A domain model is a design for a family of systems. Recognizing fundamental objects (or classes) that appear in many or all systems is central to domain modeling. A common trait of domains is that not all of its systems export the same interface. Thus, it is quite possible for two systems to export exactly same fundamental object, but disagree on the set of operations (i.e., methods) that can be performed on it.

Consider modeling a domain of textbook applications. Textbooks would clearly be fundamental objects. It seems reasonable to give textbooks the attributes `author`, `title`, and `subject`. This would be acceptable if all applications needed to distinguish textbooks on the basis of these attributes. They would not be appropriate, however, if some applications maintained stock and volume information for a warehouse (where at least the `subject` attribute is irrelevant), or if other applications only recorded the materials used in manufacturing textbooks (where `author`, `title`, and `subject` are irrelevant). Clearly, the data and operations that are encapsulated by an object will vary from application to application.

This variability of object interfaces is a consequence of *subjectivity* [28], [29], [44], [45]: when modeling software domains, objects don't have single interfaces, but are described by a family of related interfaces. The interface of an object for a given application will be subjective (i.e., application-dependent).

Subjectivity is clearly relevant to software reuse. In some sense, software is analogous to a photograph. Experiences in photography tell us that no single perspective captures all aspects of an object; every perspective exposes some features, hides others, and skews the remaining. Analogously, software encodes a particular "view" or "perspective" of an object relative to the needs of a particular application. Reusing software written for one application to build another application is possible only if the views of shared objects are compatible.

Subjectivity is also relevant to generators. Generators use one of two different ways to model families of interfaces. One way is to use multiple inheritance. Multiple inheritance elegantly expresses primitive increments of interface variation and the means to combine these primitives to define the family of interfaces that arise in a domain. However, multiple inheritance fails to adequately capture the combinatorial numbers of implementations of these interfaces; only limited families of *implementations* can be expressed [27], [9].[3]

A second approach is to ignore interface variations altogether: systems and components export "standardized" interfaces and are otherwise indistinguishable except for performance-related or feature-related metrics. While this seems restrictive, in practice it works well. Components with standardized interfaces provides an effective solution for addressing the combinatorial numbers of implementations that can arise for a given interface; it simply fails to explain interface variations that can occur. Object-oriented frameworks and abstract factory design patterns take this approach [33], [34], [18], and so too it would seem GenVoca.

A general solution to the problem of generating interface and implementation variations among software systems of a domain is needed. Although components with nonstandardized interfaces seems at odds with the GenVoca model, we explain in the following sections that this is not the case. GenVoca components have subjective (i.e., mutable) interfaces and bodies, i.e., their interfaces and bodies adjust upon instantiation to a "standard" that is system-specific (i.e., application-specific). We begin by explaining why "cast-in-concrete" interfaces cannot be part of a general solution.

4.1 The Myth of Standardized Interfaces

GenVoca components are composable because they export and import "standardized" interfaces. Yet subjectivity tells us that no single interface captures all views of an object. What then does it mean for a GenVoca interface to be "standardized?" How are operations chosen to be included in a "standardized" interface? What criteria is used to exclude operations? One could argue if GenVoca generators purport to produce high-performance software, then no operation could be excluded because that operation might be needed for performance-critical applications. Indeed, when GenVoca interfaces are defined, there are operations that most people would agree are "core" or "intrinsic," but many other operations are indeed "optional" or "subjective."

EXAMPLE. The core operations that one can perform on P2 containers are element retrievals, updates, insertions, and deletions. However, there is an infinite number of optional operations: Count the number of elements in the container, return the last element inserted, insert an element after a given element, etc. Core operations are distinguished from optional operations subjectively, i.e., by their perceived need for the target applications that P2 was initially designed to support.

The notion that standardized interfaces are immutable or cast-in-concrete in GenVoca is a myth. Each component encapsulates a domain-specific feature. For programmers or other components to take advantage of this feature, it is often necessary for a component to export noncore, component-specific operations. The ability of components to augment the set of core operations that they export and import, of course, destroys any pretense of realm interfaces being immutable or cast-in-concrete. To emphasize this point, it is quite common in GenVoca for the exported interface of a generated system to change with the addition or removal of a component.

EXAMPLE. P2 has a **`size_of`** component which maintains a count of the number of elements in a container. This count variable cannot be read by a core operation. Instead, **`size_of`** exports the nonstandard **`read_size`** operation to read the count. When **`size_of`** appears in a type equation that defines a container's implementation, **`read_size`** is added to that container's interface. If **`size_of`** is removed from the type equation, **`read_size`** is removed from the interface.

EXAMPLE. P2 has a **`timestamp`** component. It appends to every element in a container the time of its insertion. The layer-specific operation **`get_timestamp`** is added to the cursor class interface for reading element timestamps. If **`timestamp`** is removed from the container's type equation, **`get_timestamp`** disappears from the cursor interface.

To illustrate the general situation, Fig. 7a depicts three symmetric layers; each layer exports and imports the same set of core operations. (Export operations are drawn above a component; import operations are drawn below.) Note that the bottom layer has an extra left operation and the middle layer has an extra right operation; neither of these extra operations are "core." Fig. 7b shows the result of composing these layers: all layers are *automatically* extended to support *both* a left and right operation. The simplest way to understand this behavior is that in layered systems, it is common for lower layers to export operations that only they understand. For these layer-specific operations to be exported through the top of the system, they must be propagated through higher layers. By the same reasoning,

3. Multiple inheritance is very restricted class extension operator. As long as the base classes (whose definitions are to be composed by multiple inheritance) represent largely orthogonal concepts with independent implementations, multiple inheritance produces correct results. However, if base classes have dependencies (which incidentally is the norm for GenVoca components), multiple inheritance is inadequate to merge class functionalities, and a different method composition is needed.

if the middle or lower layer is removed from a composition, its layer-specific operation will be removed from all layers of that composition. It is in this way that GenVoca generators customize the interfaces of components (and their exported objects) and thus produce view-specific software. Furthermore, the ability to add new operations renders the distinction of core vs. layer-specific operations moot.

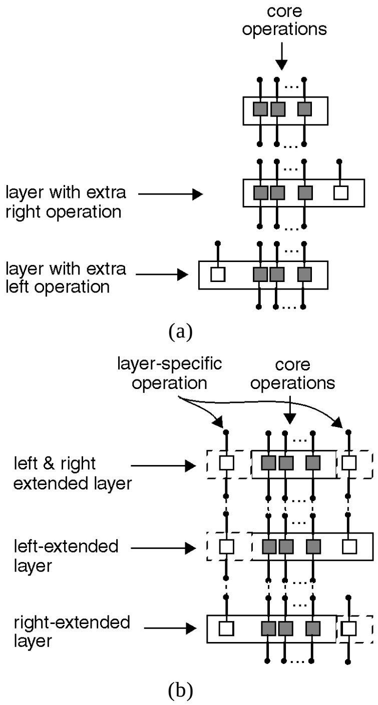

Fig. 7. Propagation of layer specific operations. (a) layers prior to composition; (b) layers after composition.

However, this does raise an interesting dilemma: On the one hand, the composibility of GenVoca components is dependent on standardized interfaces. On the other hand, individual components may export nonstandard operations. Although this seems contradictory, subjectivity offers a resolution.

GenVoca components really don't have single interfaces —their instances can export any one of a family of related interfaces. When GenVoca components are composed, their interfaces are automatically adjusted to a "standard" that is specific to that type equation (i.e., the resulting interfaces are system-specific). Fig. 7a shows that prior to composition, the top, middle, and bottom components do *not* export the same interface; yet Fig. 7b shows that after composition their instances do, and this interface is specific to this particular composition. Thus, standard interfaces do not mean cast-in-concrete in GenVoca; they are indeed subjective.

It is worth exploring how subjective interfaces are related to the conventional OO concept of inheritance (subclassing), which also supports operation propagation and refinement. Inheriting operations from superclasses is the only way operations are automatically propagated from one class definition to another in OO models. Fig. 8a shows an inheritance hierarchy of three classes, rooted at class `X`. The operations of class `Z` are those that are defined by `Z` and those that are inherited from `X` and `Y`. The direction of operation propagation is top-down (i.e., from superclasses to subclasses). In addition, one can view inheritance (subclassing) hierarchies as a composition of refinements: `X` refines some abstract interface, `Y` refines `X`'s implementation, and `Z` refines `Y`'s implementation.

Fig. 8b depicts a component instantiation hierarchy `T[M[B]]`. Although drawn deliberately like Fig. 8a, edges between boxes denote realm parameter instantiation, *not* inheritance. Like the inheritance hierarchy, `T` refines an abstract interface, `M` refines `T`'s implementation, and `B` refines `M`'s implementation. But notice that the direction in which operations are propagated is exactly the *opposite* of inheritance: the operations of `T` are those that are defined by `T` and the component-specific operations that are defined by `M` and `B`.

Merely inverting the layer hierarchy (so that `B` of Fig. 8b is on the top) is not the full answer. Recent work by Van Hilst and Notkin show that stacking layers (refinements) can be implemented by adding subclasses to existing inheritance hierarchies via parameterization [57]. Although this seems to be a key observation in unraveling the relationship between subjectivity and inheritance, it fails to explain the effects of transformations that we have encountered in P2 (and other GenVoca generators). For example, while "inversion" does make the direction of operation propagation similar to Fig. 8a, the order in which refinements are composed is reversed. Top-down refinement (i.e., starting with abstract classes and refining them by the topmost layer first and the bottommost layer last) is *crucial* to P2. In short, subjectivity is different or more general than the traditional notions of static subclassing [44], [45], [28],[29], [7].

Adding new operations to an interface is simple, but how does one automatically manufacture a method for such operations on a per-component basis? How can components with subjective interfaces be implemented? What programming language features are needed to support subjectivity? What programming paradigm unifies these ideas? In the following section, we review actual implementations of components with subjective interfaces in four *independently-conceived* GenVoca generators. Although all four solutions are outwardly different, they are fundamentally similar. Afterward, we distill the essence of these solutions, and in doing so, we answer the questions posed in this paragraph.

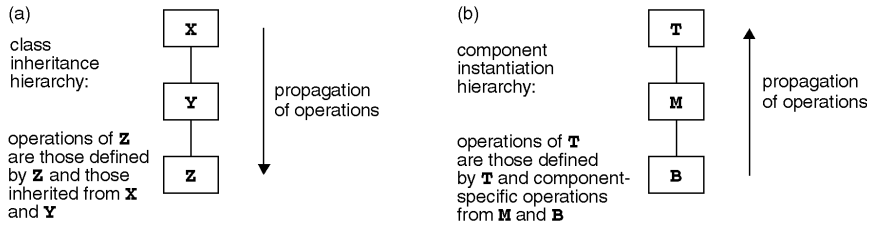

Fig. 8. Inheritance vs. subjectivity.

4.2 Four Implementations

Generators perform tasks that are automatable (e.g., code generation, composition, composition validation, optimization, etc.); the tasks that are not automatable (e.g., recognizing new domain abstractions, recognizing new components of a realm, recognizing design rules and composition constraints, understanding domain knowledge, etc.) are the responsibilities of domain analysts and component implementors. It is this perspective that one should keep in mind when reviewing the following implementations of subjective components.

Genesis. Genesis was the first GenVoca generator; it demonstrated that customized database management systems (in excess of 50,000 lines of code) could be assembled from prefabricated components [1]. Genesis relied on a rather rigid (and in hindsight) inflexible way of accommodating subjectivity; realm interfaces evolved as new components were written. That is, when a new component `K` was added to realm `R`, and `K` exported nonstandard operation `O`, all components of `R` were manually retrofitted to export `O`. This did not mean that every component of `R` had to implement `O`; nonstubbed implementations were provided only for those components where it made sense to do so.

Thus, the interfaces of Genesis components were adjusted manually whenever a new component was added to a realm.[4] There was no subsequent adjustment of interfaces if type equations did (or did not) use a particular component. This approach worked because of the objective of Genesis, namely, to demonstrate DBMS synthesis. Performance wasn't an issue and a large user community (that would insist on having many optional operations) was not envisioned.[5]

Avoca. Avoca / *x*-kernel demonstrated that highly layered communications protocols could be more efficient and more extensible than monolithic protocols [32], [1]. Avoca realm interfaces were rigid (i.e., cast-in-concrete) sets of operations. *Microprotocols*, the name given to Avoca components, implemented a fixed-set of core operations for transmitting messages and opening and closing sessions, plus an additional operation `control`. Every microprotocol could export zero or more control functions—what we have called layer-specific operations—that only it understood. Calls to these functions were made through `control` which took a pair of arguments: a control function name and a pointer to the control function's argument list. A `control` operation was implemented as a switch statement; there was one case for each of the microprotocol's control functions and a default case for transmitting the control operation to the next lower microprotocol:

```
void control( int op_id, arg *arg_list )
{
   switch( op_id )
   {
      case op1: // code for layer-specific
                   operation #1
      case op2: // code for layer-specific
                   operation #2
      ...
      default : // call control operation of
                   lower layer lower.control
                   (op_id, arg_list);
   }
}
```

The advantage of this approach is its generality; it can accommodate any number of control functions per microprotocol and it does not require component interfaces to be modified with the addition or removal of a layer-specific operation.[6] The drawbacks are program clarity and performance. Coding function calls via switch statements and marshalling arguments are well-known to be obscure ways of programming [33]. Moreover, there can be a considerable performance overhead in processing control operations. Calling a control function essentially requires polling each component of a type equation to test if it could process the function. Control functions were not called frequently enough in Avoca for their inefficiencies to be problematic.

Ficus. Ficus builds customized file systems from a single realm of components [30]. All Ficus layers support the same set of core operations plus any number of layer-specific operations. The reliance of Ficus on the Unix vnode facility encouraged a uniform treatment of core and layer-specific operations. It also encouraged the interface of a file system to be determined at configuration time, where every layer of its type equation is polled for the set of operations that it implements. The union of all operations from all layers in a file system defines the interface to that file system. All layers of that file system are then automatically extended to support this interface. Since it is not possible to anticipate what operations would be provided by other (possibly yet-to-be-written) layers, every Ficus layer provides a `bypass` method for unanticipated operations. Usually, the default method is simply to transmit calls of unanticipated operations to the next lower layer. However, nondefault methods do arise.

An example of a nondefault method occurs in protection layers. Protection layers validate access privileges of clients prior to performing file operations. The bypass method for unanticipated operations is to verify the user's ability to access the given file. Variations on this theme (e.g., testing for read-only access or write access) are possible [30], [31].

P2. A P2 layer is a transformation between the layer's export interface and its import interface(s); only layer-specific operations and core operations for which nonidentity transforms are performed need to be defined. When the P2 generator is compiled, the union of the export interfaces of every layer in a realm is determined. Each layer is then automatically extended to support this union interface. Operations that are undefined by a layer are (in effect) supplied default bodies which transmit the operation to the next lower layer. Default methods can be overridden on a per-class basis.

4. Components were added to realms in the order that maximally stressed realm interfaces. We discovered that once the first few components were added, realm interfaces quickly reached a steady state. So backtrackingh and global updating was infrequent.

5. A consequence of this approach was the need for design rules: Although the interfaces of all components of realm `R` were syntactically identical, not all components implemented operation `O`. This meant that components of `R` were not always interchangeable and that not all syntactically correct compositions of Genesis components were semantically correct. Design rule checking was needed to validate compositions.

6. Note that a nondefault method, i.e., something other than transmitting a control function call to lower layers, could easily be encoded in this scheme.

A P2 component that has multiple nondefault methods is monitor, which encapsulates the transformation that converts a container into a monitor; i.e., all accesses to the container occur within a critical region. monitor exports two classes: container and cursor. The monitor rewrite adds a semaphore data member sem to the container class and modifies the methods of all cursor and container operations by wrapping them with wait and signal calls.

Sketches of the monitor operation rewrites are shown below. container_op pattern-matches with any container operation and "…" is bound to its arguments. The rewritten method is enclosed within braces { }: a wait is performed, then the actual operation itself is processed (by the layer immediately beneath monitor), followed by a signal:

```
container_op ( … )
{ sem.wait();
  lower_container.container_op( … );
  sem.signal();
}
```

The rewrite of cursor operations is different (albeit slightly) from that of container operations: the container semaphore must be accessed indirectly:

```
cursor_op( … )
{ container->sem.wait();
  lower_cursor.cursor_op( … );
  container->sem.signal();
}
```

In general, a bypass method is specified for each class that is exported by a component. It is not difficult to imagine that even finer granularities of rewrites may be needed.[7]

4.3 A Model of Subjectivity

Although different, there are striking commonalities in the subjectivity mechanisms of the Genesis, Avoca, Ficus, and P2 generators. In this section, we propose a model of these mechanisms as extensions to the P++ language [50], [51]. P++ is a superset of C++ that is specifically designed to support the GenVoca model. Among its extensions are declarations for realms, components, and parameters. The current version of P++ permits the composition of components at compile-time; it does not yet support run-time compositions or the concept of subjectivity discussed in this paper. (Realm interfaces are standardized manually at design-time, much like component interfaces were standardized in Genesis.) Our proposed extensions to P++ have been implemented in the P2 generator, so we will be describing an abstraction of a working system. Our choice of P++ as the medium of explanation stems from the recognition that language support for a design paradigm greatly simplifies the application and understanding of that paradigm.

As a running example, we will use the container data structure abstraction of P2 [3]. This abstraction is represented by three classes: elements, containers, and cursors. Elements are the objects stored in containers. Cursors are used to retrieve and update objects within containers.

7. As an example, if an operation only reads a private data member of a class, there should be no need to execute the read within a critical region. Thus, the wrapping of wait and signal operations around a method could be selective.

Realms. A realm interface defines a programming interface for a domain abstraction. It is a specification of the prototypes of one or more classes and functions; realms have no variables or data members. The DS (container data structures) realm is shown in Fig. 9a. DS consists of two classes, container and cursor, that are parameterized by a third class e, the class of elements that are to be stored in containers and that are to be accessed by cursors.

(a)
```
template <class e>
realm DS
{
   class container
   { container ();
     bool is_full();
     ... // other operations
   };

   class cursor
   { cursor (container *c);
     void advance ();
     e* insert ( e *obj );
     void remove ();
     ... // other operations
   };
};
```

(b)
```
template <class e>
realm DS_size : DS< e >
{
   class container { int read_size(); };
};

template <class e>
realm DS_time : DS< e >
{
   class cursor { int get_timestamp();  };
};
```

(c)
```
template <class e>
realm DS_size_time : DS_size<e>, DS_time<e>;
```

Fig. 9. Realm and subrealm declarations.

To support subjectivity and interface variations, we introduce subrealms to P++, i.e., specializations/subtypes of a realm definition. Fig. 9b shows two subrealms of DS. DS_size extends the container class with the read_size operation and DS_time extends the cursor class with the get_timestamp operation. Note that the parameter(s) of superrealms are inherited by their subrealms (i.e., DS is parameterized by class e, thus e is a parameter of subrealms DS_size and DS_time). Fig. 9c shows an alternative way of defining subrealms as a union of previously declared subrealms.

Components. A P++ component is a *large-scale refinement* of its realm interface. It is defined as a set of consistent data refinements, nonbypass operation refinements, and bypass refinements. The size_of component is shown in Fig. 10a. size_of refines the container class by adding the variables lower and count, and explicitly refining the constructor and read_size operations. All other container operations are implicitly refined by the container bypass. size_of refines the cursor class by adding the variables

(a)
```
template <class e, DS<e> x>
component size_of: DS_size< e >
{
  class container
  { friend class cursor;
    x::container lower;
    int          count;

    container()    { count = 0; };
    int read_size(){ return count; };

    bypass_type bypass(bypass_args)
    { return lower.bypass(bypass_args); };
  };

  class cursor
  {  x::cursor *lower;
     container *c;

     cursor( container *k )
     {  c = k;
        lower = new x::cursor(&(c->lower)); };

     e* insert( e *element )
     {  c->count++;
        return lower->insert(element);  };

     void remove()
     {  c->count--;
        lower->remove();  };

     bypass_type bypass(bypass_args)
     { return lower->bypass(bypass_args); };
  };
};
```

(b)
```
template < class e, DS<e> x >
component monitor: DS< e >
{
  class container
  { friend class cursor;
    x::container lower;
    semaphore    sem;

    container() { };

    bypass_type bypass(bypass_args)
    { bypass_type tmp;
      sem.wait();
      tmp = lower.bypass(bypass_args);
      sem.signal();
      return tmp;  }
  };

  class cursor
  { x::cursor *lower;
    container *c;

    cursor( container *k )
    { c = k;
      lower = new x::cursor(&(c->lower)); };

    bypass_type bypass(bypass_args)
    { bypass_type tmp;
      c->sem.wait();
      tmp = lower->bypass(bypass_args);
      c->sem.signal();
      return tmp;  }
  };
};
```

Fig. 10. The `size_of` and `monitor` components.

`lower` and `c`, plus explicit refinements of the constructor, `insert` and `remove` operations (that increment and decrement `count`). All other `cursor` operations are implicitly refined by the cursor bypass. There are three points about this example that we want to elaborate.

First, rewrites of unspecified operations are expressed by the P++ `bypass` construct. `bypass` pattern-matches with the name of any operation that is not explicitly declared within the enclosing class but is an operation that is to be exported by that class. `bypass_type` is the return type of that operation and `bypass_args` matches its argument list. The body of `bypass` defines the method rewrite. For example, the `size_of` bypasses for both `cursor` and `container` transmit the operation verbatim to the layer immediately beneath `size_of`. Fig. 10b shows the `monitor` component which does not use verbatim bypasses.

Second, bypasses complicate type checking in P++ because they allow interfaces of component instances to be of an arbitrary size. Consequently, component instances can have varying realm export and import types. To type check component definitions, we must ensure that the type signatures of the realm operations that are explicitly referenced in the component body match those of the export and import realms. For example, `size_of` explicitly exports the `insert`, `remove`, `read_size` and constructor operations; their signatures are covered by the `DS_size` realm. (These signatures could also be covered by `DS_size_time` and many other larger realms; `DS_size` is the smallest cover given the realms of Fig. 9). Further, `size_of` explicitly imports the `insert`, `remove` and constructor operations; their signatures are covered by the `DS` realm. Thus, the `size_of` component is declared to minimally export the realm `DS_size` <e> and to minimally import `DS<e>`.

Third, an implicit assumption of the `DS` abstraction is that the only way elements can be added or removed from containers is via the cursor operations `insert` and `remove`. Should a new layer `L` introduce another operation for adding or removing elements, the `size_of` component may not maintain an accurate count of the number of elements in a container. This means that `size_of` cannot be composed with `L` to yield a valid type equation. Such a constraint can be expressed using design rules. Alternatively, `size_of` could be made compatible with `L` if it explicitly defines rewrites for all element addition and removal operations of `L`. As mentioned in Section 4.2, the recognition of the incompatibility of component compositions (or the modification of components to make them consistent) is borne by domain analysts and component implementors, and is not done automatically by generators.

Type Equations. Components are composed in P++ in `typedef` declarations. Suppose `array` and `avl` are components that implement the `DS` interface and do not export layer-specific operations. Type equations `C1` and `C2` (below) will generate systems that export the `DS_size` interface:

```
typedef size_of[avl]   C1;
typedef size_of[array] C2;
```

```
main()
{  DS_size::container *cont;
   DS_size::cursor    *curs;

   if (environment_variable)
   {  cont = new C1::container;
      curs = new C1::cursor;   }
   else
   {  cont = new C2::container;
      curs = new C2::cursor;  };
   ...
}
```

Fig. 11. Environment-selectable implementations.

Given these declarations, the program of Fig. 11 is type correct. An environment variable decides whether container and cursor implementations of type `C1` or `C2` should be used during program execution.

Now suppose `avl` and `array` are modified to export layer-specific operations: `avl` additionally exports the `num_balances` operation, while `array` additionally exports the `num_free_slots` operation. As explained in Section 4.1, the compositions `C1` and `C2` will generate different systems, both of which have slightly different interfaces than `DS_size`. `C1` would export the `DS` core, `num_balances`, and `read_size` operations, while `C2` would export `DS` core, `read_size`, and `num_free_slots`. Note that the program of Fig. 11 would no longer be type correct (as `C1`, `C2`, and `DS_size` are distinct types—they don't have identical signatures and are not explicitly related in a subtyping hierarchy), and will fail to compile. This, despite the fact that the additional operations that were generated, `num_free_slots` and `num_balances`, are never referenced.

The problem is that `C1` and `C2` have manufactured interfaces that don't match any explicitly defined realm. For an application to insulate itself from irrelevant operations of components, it must use a realm declaration that defines the interface that all generated systems should export. This could be accomplished by *casting* type equations to yield the subjective view that is required:

```
typedef (DS_size) size_of[avl]   C1;
typedef (DS_size) size_of[array] C2;
```

That is, our application interacts with generated subsystems via interface `DS_size`. `C1` and `C2` are now equations that define different systems that implement `DS_size`. Hence, instances of `C1` and `C2` are plug-compatible and thus the program of Fig. 11 is now type correct. From the perspective of the P++ compiler, casting may actually simplify the composition of components. Once the export interface of a generated system is known, operations that do not belong to this interface need not be generated.

Open Problems. The proposed extensions to P++ take us closer to a better understanding of programming language support for GenVoca and components with subjective interfaces. However, several important open problems remain. P++ components are presently composable only at application compile-time; ideally, components should also be composable at run-time. Such a capability would permit software systems to evolve dynamically. Although there are several possibilities on how to proceed (e.g., [17], [30], [32]), it is not yet clear what run-time capabilities should be added to P++ to support the dynamic composition of components with bypass methods.

Another challenging problem is how to encapsulate design rules within P++ components. Presently, design rule checking is accomplished with a tool external to P++ (e.g., `dreck`). Thus, design rules for components are specified separately from P++ component definitions. The difficulty of integration is that design rules would extend the P++ type checking system, thereby requiring P++ to be a fairly open compiler. Once again, there are possibilities on how to proceed (e.g., [45]).

4.4 Related Work on Subjectivity

Frameworks. An object-oriented *framework* is a set of abstract classes with their own sets of concrete classes. The combinations of concrete classes that can work together can be defined in a variety of ways (e.g., informally or using factory design patterns [18]); there is no fixed rule about how concrete classes can be paired. Realms and frameworks are indeed similar [1]: the *n* classes of a realm's interface correspond to the *n* abstract classes of a framework. Each GenVoca/P++ component specifies an *n*-tuple of concrete classes (one concrete class per abstract class) that work together as a unit. The differences between realms and frameworks are: 1) the subjective nature of component interfaces and 2) the need for bypass methods to encapsulate the operation refinements of components.

Subjectivity. Subjectivity arose from the need for simplifying programming abstractions, e.g., defining views that emphasize relevant aspects of objects and that hide irrelevant details, [26], [18]. This led to a connection of object modeling with view integration in databases, namely, object models can be defined as a result of integrating different application views of objects [23], [27]. Ossher and Harrison took an important step further by recognizing that application-specific views of inheritance hierarchies can be produced automatically by composing "building blocks" called extensions [44]. An *extension* encapsulates a primitive aspect or "view" of a hierarchy, whose implementation requires a set of additions (e.g., new data and method members) to one or more classes of the hierarchy. A customized "view" of an inheritance hierarchy could therefore be defined by composing extensions. Extensions and their compositions are similar to the GenVoca concepts of components and type equations. Moreover, similar scalability arguments have been advanced independently for both models and that not all compositions of extensions (or GenVoca components) may be semantically correct (c.f., [3] and [6]). The models are not the same, however, as (for example) extensions have no counterparts to realms and realm parameters.

It is worth noting that a rather different and powerful approach to views and software reuse has been proposed by Goguen [21], [22] and Novak [43]. The essential idea is to define a customized interface to an object (or sets of objects); a view defines a mapping of each object to its customized interface.

Module Interconnection Languages (MILs). Limited forms of subjectivity can be achieved through MILs. Microsoft's Common Object Model (COM) permits objects to have a set of (upwards compatible) interfaces to maintain backwards compatibility with old views of objects. As another example, Goguen's model of parameterized programming (LIL) permits simple transforms on modules, such as combining modules by merging their operations and types; types, operations, and exceptions can be added, exchanged, removed, or renamed, etc. [21], [22], [55]. While the basic transforms are present to achieve subjectivity, there are no higher-order transforms that query module interfaces, wrap all or selected operations of a module, and propagate operations to other modules automatically; such capabilities can only be specified manually on a per-module basis.

Reflectivity. Bypass methods correspond to *method wrappers* or *before* and *after* methods in metaobject protocols [36]. CLOS was among the first languages to have method wrappers. Wrappers in CLOS are different than in P++ as they are defined on a per-operation basis. A model of wrappers that is closer to P++ is that of SOM metaclasses, where all (or selected) operations of a class can be wrapped by before and after methods [15], [17]. Wrappers are defined in SOM by overriding the dispatch methods of metaclasses. Thus, to define the equivalent of the P++ `monitor` component would require four separate definitions in SOM: two classes (`cursor` and `container`) and two metaclasses (a metaclass for wrapping `cursor` operations and a metaclass for wrapping `container` operations). SOM has no mechanism to encapsulate multiple classes and metaclasses. In contrast, the P++ `component` construct allows multiple classes to be encapsulated and does not require the need for metaclasses to specify wrappers. A more important distinction is that wrappers are composed in SOM (and CLOS) through class inheritance; wrappers (bypass methods) are composed in P++ through realm parameter instantiation. Thus, the mechanisms for wrapper composition in both models are different.

5 Conclusions

Software system generators will become important tools for software developers. An important class of generators, called GenVoca generators, utilize libraries of reusable components to assemble complex, high-performance systems quickly and cheaply. In this paper, we have presented solutions to two fundamental problems of GenVoca generators: validating component compositions and manufacturing subjective interfaces for component instances.

First, every library component has limitations, called design rules, on how it can be combined with other components. Experience has shown that validating component compositions by casual inspection is error-prone; as the number of components and the complexity of their rules grow, a mechanical approach to validation is absolutely essential. We have developed domain-independent algorithms that rely on shallow consistency checking to validate component compositions. Experience confirms that domain-specific instances of our algorithms are practical: they are simple, easy to implement, and efficient. More-over, they offer powerful explanation-based error reporting capabilities to suggest to users how incorrect compositions can be repaired.

We also observed that the number primitive predicates that are needed for design rule checking is surprisingly small. We believe the explanation for this lies in the power of standardizing domain abstractions and their programming interfaces (i.e., the core of GenVoca) to control the complexity of families of software systems. Components that are designed to be interoperable, plug-compatible, and interchangeable often make otherwise difficult problems tractable.

Second, we explored an unusual feature of GenVoca components. Unlike traditional software modules whose interfaces remain unchanged upon instantiation, GenVoca components mutate upon instantiation—their interfaces and bodies enlarge automatically to meet interface requirements that are imposed by a system. The mutability of interfaces is interesting in the context of GenVoca because the composibility of components is based on components exporting and importing standardized interfaces.

We have shown that standardized interfaces and mutable interfaces are not inconsistent. The principle of subjectivity asserts that when modeling a domain of applications, objects do not have single interfaces, but rather are described by a family of related interfaces. At component instantiation time, an interface is manufactured for each object/class of a component that is appropriate to the system in which it is to be used. Thus, all components in a system that export or import these objects/classes must use this system-specific standard. It is in this way that the interfaces of GenVoca components are automatically customized. We outlined linguistic extensions to C++ that would support components with subjective interfaces.

So that others may learn from our work, `dreck`, and P2 are available free of charge via the Predator web page: `http://www.cs.utexas.edu/users/schwartz/`.

Acknowledgments

We thank Dewayne Perry for stimulating discussions on Inscape and drafts of the design rule checking sections. We also thank Ira Baxter, Paul Clements, Dave Weiss, Chris Lengauer, Bruce Weide, and Steve Edwards for their insights on design rule checking. We are grateful to Mike Hewett and Chris Petrock for their CS395T term project that gave rise to our work.

We thank Reed Little (SEI) for pointing out the similarity of method wrapper mechanisms in CLOS and FLAVORS to the operation bypasses in GenVoca components. We also thank Lance Tokuda, Vivek Singhal, Trudy Levine, Peter Clark, Jeff Thomas, and Guillermo Jimenez-Perez for their useful comments on earlier drafts of the subjectivity sections of this paper. Finally, we thank Ira Baxter for his insightful comments on revisions and on the helpful comments of anonymous referees.

This work was supported in part by Microsoft Research, Schlumberger, the University of Texas Applied Research Labs, and the U.S. Department of Defense Advanced Research Projects Agency in cooperation with the U.S. Wright

Laboratory Avionics Directorate under Contract No. F33615-91C-1788.

REFERENCES

[1] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM TOSEM*, pp. 355-398, Oct. 1992.

[2] D.S. Batory and J.R. Barnett, "DaTE: The Genesis DBMS Software Layout Editor," *Conceptual Modeling, Databases, and CASE*, P. Loucopoulos and R. Zicari, eds., pp. 201-222, John Wiley & Sons, 1992.

[3] D. Batory et al., "Scalable Software Libraries," *Proc. ACM SIGSOFT*, pp. 191-199, Dec. 1993.

[4] D. Batory and D. Vasavada, "Software Components for Object-Oriented Database Systems," *Int'l J. Software Eng. and Knowledge Eng.*, pp. 165-192, 1993.

[5] D. Batory et al., "The GenVoca Model of Software-System Generators" *IEEE Software*, pp. 89-94, Sept. 1994.

[6] D. Batory and B.J. Geraci, "Validating Component Compositions in Software System Generators," UT/CS TR-95-03, Univ. of Texas at Austin, 1995.

[7] D. Batory, "Inheritance and Subjectivity," to appear.

[8] I. Baxter, "Design Maintenance Systems, *Comm. ACM*, pp. 73-89, Apr. 1992.

[9] T. Biggerstaff, "The Library Scaling Problem and the Limits of Concrete Component Reuse," *Int'l Conf. Software Reuse*, Rio de Janeiro, pp. 102-109, 1994.

[10] L. Blaine and A. Goldberg, "DTRE—A Semi-Automatic Transformation System," *Constructing Programs from Specifications*. Elsevier Science, 1991.

[11] G. Booch, *Object-Oriented Design with Applications*. Benjamin-Cummings, 1991.

[12] C. Chambers and G.T. Leavens, "Type Checking and Modules for Multi-Methods," *OOPSLA '94*, pp. 1-15, 1994.

[13] L. Coglianese and R. Szymanski, "DSSA-ADAGE: An Environment for Architecture-Based Avionics Development," *Proc. AGARD*, 1993.

[14] S. Cohen et al., "Models for Domains and Architectures: A Prescription for Systematic Software Reuse," *AIAA Computing in Aerospace*, 1995.

[15] S. Danforth and I. Forman, "Reflections on Metaclass Programming in SOM," *OOPSLA '94*, pp. 440-452, 1994.

[16] E.W. Dijkstra, "The Structure of THE Multiprogramming System," *Comm. ACM*, pp. 341-346, May 1968.

[17] I.R. Forman, S. Danforth, and H. Madduri, "Composition of Before/After Metaclasses in SOM," *OOPSLA '94*, pp. 427-439, 1994.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[19] D. Garlan et al., "Exploiting Style in Architectural Design Environments," *ACM SIGSOFT*, pp. 175-188, 1994.

[20] D. Garlan et al., "Architectural Mismatch or Why It's Hard to Build Systems Out of Existing Parts," *Int'l Conf. Software Eng.*, 1995.

[21] J.A. Goguen, "Reusing and Interconnecting Software Components," *Computer*, Feb. 1986.

[22] J.A. Goguen, "Parameterized Programming and Software Architecture," *Int'l Conf. Software Reuse*, pp. 2-10, 1996.

[23] P. Goldstein et al., "An Experimental Description-Based Programming Environment: Four Reports," Technical Report CSL-81-3, Xerox PARC, Mar. 1981.

[24] H. Gomaa et al., "A Prototype Domain Modeling Environment for Reusable Software Architectures," *Int'l Conf. Software Reuse*, pp. 74-83, 1994.

[25] M.L. Griss and K.D. Wentzel, "Hybrid Domain-Specific Kits for a Flexible Software Factory," *ACM SAC '94*, Mar. 1994.

[26] B. Hailpern and H. Ossher, "Extending Objects to Support Multiple Interfaces and Access Control," *IEEE Trans. Software Eng.*, pp. 1,247-1,257, Nov. 1990.

[27] W. Harrison et al., "Integrating Coarse-Grained and Fine-Grained Tool Integration," *Workshop on Computer-Aided Software Eng.*, July 1992.

[28] W. Harrison and H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)," *OOPSLA '93*, pp. 411-427, 1993.

[29] W. Harrison, H. Ossher, R.B. Smith, and D. Ungar, "Subjectivity in Object-Oriented Systems: Workshop Summary," Addendum to *OOPSLA '94*, 1994.

[30] J. Heidemann and G. Popek, "File System Development with Stackable Layers," *ACM TOCS*, Mar. 1993.

[31] J. Heidemann, e-mail correspondence, 1995.

[32] N. Hutchinson and L. Peterson, "The x-kernel: An Architecture for Implementing Network Protocols," *IEEE Trans. Software Eng.*, pp. 64-76, Jan. 1991.

[33] R.E. Johnson and B. Foote, "Designing Reusable Classes," *J. Object-Oriented Programming*, June/July 1988.

[34] R.E. Johnson, "Documenting Frameworks Using Patterns," *OOPSLA '92*, pp. 63-76, 1992.

[35] M.D. Katz and D.J. Volper, "Constraint Propagation in Software Libraries of Transformation Systems," *Int'l J. Software Eng. and Knowledge Eng.*, vol. 2, no. 3, 1992.

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

[37] J.C.S. do Prado Leite et al., "Draco-PUC: A Technology Assembly for Domain-Oriented Software Development," *Int'l Conf. Software Reuse*, pp. 94-101, 1994.

[38] D. McAllester, "Variational Attribute Grammars for Computer Aided Design," ADAGE-MIT-94-01, 1994.

[39] M. Moriconi and X. Qian, "Correctness and Composition of Software Architectures," *ACM SIGSOFT '94*, 1994.

[40] J. Neighbors, "Software Construction Using Components," PhD thesis, ICS-TR-160, Univ. of California at Irvine, 1980.

[41] M. Nenninger and F. Nickl, "Implementing Data Structures by Composition of Reusable Components: A Formal Approach," *Proc. ICSE-17 Workshop Formal Methods Applications in Software Eng. Practice*, Apr. 1995.

[42] J.Q. Ning et al., "An Architecture-Driven, Business-Specific, and Component-Based Approach to Software Engineering," *Int'l Conf. Software Reuse*, pp. 164-174, 1994.

[43] G.S. Novak, "Creation of Views for Reuse of Software with Different Data Representations," *IEEE Trans. Software Eng.*, pp. 993-1,005, Dec. 1995.

[44] H. Ossher and W. Harrison, "Combination of Inheritance Hierarchies," *OOPSLA '92*, pp. 25-40, 1992.

[45] H. Ossher et al., "Subject-Oriented Composition Rules," *OOPSLA '95*, pp. 235-250, 1995.

[46] D.E. Perry, "Software Interconnection Models," *Int'l Conf. Software Eng.*, 1987.

[47] D.E. Perry, "The Logic of Propagation in the Inscape Environment," *ACM SIGSOFT*, 1989.

[48] D.E. Perry, "The Inscape Environment," *Int'l Conf. Software Eng.* 1989.

[49] D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Eng. Notes*, Oct. 1992.

[50] V. Singhal and D. Batory, "P++: A Language for Large-Scale Reusable Software Components," WISR, Owego, New York, Nov. 1993.

[51] V. Singhal, "A Programming Language for Writing Domain-Specific Software System Generators," PhD thesis, Dept. of Computer Sciences, Univ. of Texas at Austin, Aug. 1996.

[52] M. Sitaraman and B. Weide, "Component-Based Software Using RESOLVE," *ACM Software Eng. Notes*, Oct. 1994.

[53] S. Sreerama, D. Fleming, and M. Sitaraman, "Graceful Object-Based Performance Evolution," *Software Practice and Experience*, vol. 27, no. 1, pp. 111-122, Jan. 1997.

[54] Symbolics, Inc., Intermediate Lisp Programming, Sept. 1984.

[55] W. Tracz, "LILEANNA: A Parameterized Programming Language," *Proc. Int'l Conf. Software Reuse*, pp. 66-80, 1993.

[56] J. Udell, "Componentware," *BYTE*, May 1994.

[57] M. Van Hilst and D. Notkin, "Using Role Components to Implement Collaboration-Based Designs," *OOPSLA '96*, pp. 359-369, 1996.

[58] D.M. Weiss, "Synthesis Operational Scenarios," Technical Report 90038-N. Version 1.00.01, Software Productivity Consortium, Aug. 1990.