Department of Computer Sciences
University of Texas at Austin
Technical Report TR 95-03

Validating Component Compositions in Software System Generators[1]

Don Batory and Bart J. Geraci
Department of Computer Sciences
The University of Texas
Austin, Texas 78712

Abstract

Generators synthesize software systems by composing components from reuse libraries. In general, not all syntactically correct compositions are semantically correct. In this paper, we present domain-independent algorithms for the GenVoca model of software system generation to validate component compositions. Our work relies on attribute grammars and offers powerful debugging capabilities with explanation-based error reporting. We illustrate our approach by showing how component compositions are debugged by a GenVoca generator for container data structures.

Keywords: Inscape, software architectures, software system generators, attribute grammars, domain models, GenVoca, software components, explanation-based error reporting.

1 Introduction

Years from now, much of the software that we are developing today and the software design and implementation problems that we are now addressing will be so well understood, that it will be possible to automate the development of this software for large families of applications. This will be accomplished through the use of software system generators. Such generators will automatically transform compact, high-level specifications of target systems into actual source code, and will rely on libraries of parameterized, plug-compatible, and reusable components for code synthesis.

Generators [Bat92, Bax92, Bla91, Gom94, Gra92, Lei94, Nin94] are among a number of approaches that are now 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 [Ude94]; CORBA can integrate components that are independently designed and stand-alone modules or executables in a heterogeneous environment. In contrast, generators are closer to toolkits [Gri94], object-oriented frameworks [Joh92], and other reuse-driven approaches (e.g, [Wei90, Sit94]), because they focus on software domains whose components are not stand-alone, 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 [Bat92a], is distinguished from the above approaches in that their components are parameterized forward-refinement program transformations that encapsulate consistent data and operation refinements. Components also encapsulate logic to automate domain-specific decisions about when to use a particular algorithm or when to apply a domain-specific optimization. For many domains, such decisions are essential for generating efficient code.

A problem that is fundamental to all component-based software development technologies is: does a given 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.

1. This research was supported by Applied Research Laboratories at the University of Texas and Schlumberger.

In this paper, we present domain-independent algorithms for solving the problem of design rule checking for GenVoca generators, and present the domain-specific variants that we have used in the Genesis and P2 projects. Our work is related to Perry's Inscape environment, which (among other topics) dealt with consistency checking in software composition models [Per87-89b]. 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.

2 The GenVoca Model

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 a 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* [Bat93, Big94]: 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 astronomically. Generators that use GenVoca organizations have been built for the domains of avionics, data structures, databases, file systems, and network protocols [Cog93, Bat93, Hei93, Hut91].

Components and Realms. A hierarchical software system is defined by a series of progressively more abstract virtual machines. A *component* or *layer* is an implementation of a virtual machine. The set of all components that implement the same virtual machine is called a *realm*; effectively, a realm is a library of plug-compatible and interchangeable components. In Figure 1a, realms **`S`** and **`T`** have three components, whereas realm **`W`** has four.

```
(a)  S = { a, b, c }                      (b)  S :=  a  |  b  |  c  ;

     T = { d[S], e[S], f[S] }                  T :=  d S | e S | f S ;

     W = { n[W], m[W], p, q[T,S] }             W :=  n W | m W |  p  | q T S ;
```

Figure 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`**.[2] This means that every component of **`T`** exports the virtual machine interface of **`T`** and imports the virtual machine interface of **`S`**. Thus, each **`T`** component encapsulates a complex mapping or *transformation* between the virtual machines **`T`** and **`S`**. Stated another way, each component of **`T`** implements the **`T`** abstraction; all implementations of **`T`** (in realm **`T`**) are expressed in terms of **`S`** abstractions. Similarly, components that have no (realm) parameters are terminals; components with multiple parameters (e.g., **`q[T,S]`**) simply means that the exported abstraction (of realm **`W`**) is implemented in terms of multiple abstractions (e.g., of realms **`T`** and **`S`**).

2. Parameterizations that we examine in this paper are simple enough to dispense with formal parameter names.

Systems and Type Equations. A software *system* is modeled by a 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 component 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).[3]

Grammars, Families of Systems, and Scalability. Realms and their components define a grammar whose sentences are software systems. Figure 1a enumerated realms **`S`**, **`T`**, and **`W`**; the corresponding grammar is shown in Figure 1b. Just as the set of all sentences defines a language, the set of all component compositions defines a Parnas *family of systems* [Par76]. Adding a new component to a realm is akin to adding a new rule to a grammar; the family of systems enlarges automatically. Because large families of systems can be built using relatively 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.[4] In realm **`W`** of Figure 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 semantics, performance, and behavior of the resulting system.

Design Rules and Domain Models. In principle, any component of realm **`S`** can instantiate the parameter of any component of realm **`T`**. The resulting equations would be *type correct*. Although an equation may be type correct, there are always certain combinations of components that are semantically incorrect. That is, there are often domain-specific constraints *in addition to* implementing a particular virtual machine that 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 *reference architecture model* (or *domain model*) for a GenVoca generator consists of realms of components and design rules that govern component composition. In the next section, we briefly review the domain model of the P2 generator and illustrate some of its design rules.

3 P2 Domain Model

P2 is a GenVoca generator for container data structures [Bat93-94]. The domain model of P2 relies on two realms: **`ds`** and **`mem`**. **`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 cursor-container mappings (e.g., free lists of previously deleted elements, sequential

3. Note that composing components can be interpreted as stacking layers in hierarchical software systems. We use the terms component and layer interchangeably in this paper.

4. Unix file filters can be composed in arbitrary orders and are simple examples of symmetric components. Other examples are given in [Bat92a].

and random storage). **mem** components export standardized memory allocation and deallocation operations. Among its members are components that manage space in persistent and transient memory.

```
ds  = { bintree[ ds ],     // binary tree
        dlist[ ds ],       // unordered doubly-linked list
        odlist[ ds ],      // key-ordered doubly-linked list
        avail[ ds ],       // free list of deleted elements
        mlist[ ds, ds ],   // multilist indexing
        malloc[ mem ],     // heap storage
        array[ mem ],      // sequential storage
        inbetween[ ds ],   // has deletion actions for some components
        top2ds[ ds ],      // the topmost layer of a ds expression
        … }

mem = { transient,         // transient memory allocation
        persistent,        // memory mapped persistence
        … }
```

Currently there are over fifty components in P2, most of which are symmetric. Container data structures are defined by type equations that typically reference from five to twenty components. Unfortunately, the correctness of even the simplest equations is not obvious. Validation is complicated by the fact that many components have nonobvious rules for their use.

As a simple example, the **inbetween** component encapsulates algorithms that are common to many data structure components (e.g., **bintree** and **dlist**). These algorithms deal with the positioning of a cursor immediately after an element has been deleted (e.g., does the cursor point to a "hole" 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 (i.e., factored) 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 (**dlist**, **bintree**, etc.) and it should precede *all* such components in the equation. The **right** equation, below, shows a correct usage — i.e., **inbetween** precedes all data structure components. The **wrong** equation, below, shows an incorrect usage: a data structure component **dlist** appears prior to **inbetween**.

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

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 tremendous burden from P2 users. We first present a general model of design rule checking in Section 4 and then show how we adapted the model to P2 and Genesis generators in Section 5 and Section 6.

4 A Model of Design Rule Checking

Perry's Inscape is an environment for managing the evolution of software systems [Per87-89b]. 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 the system that uses the 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 states of software system (or type equation) development; it does *not* model states of system execution. Figure 2 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 instantiation, **`State`** has the value **`has-loops`** (meaning that **`s'`** has loops). *Design rule checking deals with the testing and assignment of system design states; it assumes that all transformations (components) are semantically correct.*

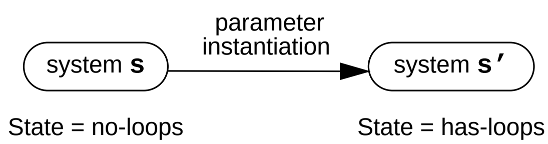

Figure 2: Modeling States of Program Development

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.[5] 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.[6]

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.

> **Example**. Suppose component **`k`** has a precondition that attribute **`A`** must have the value **`v`** (see Figure 3a). 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**.

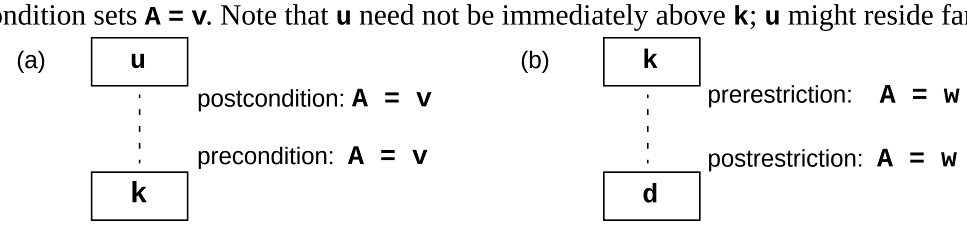

Figure 3: Different Kinds of Design Rules

5. We use the terms “higher” and “lower” refer to relative positions of components within a type equation when the equation is interpreted as a (possibly nonlinear) stack of layers. The outermost component of an equation is the “highest” component, and the innermost components are the “lowest”.

6. There may be some dispute on the proper terminology to use; preconditions and postconditions usually refer to run-time properties, not design-time properties. As there seems to be no commonly used terms for design-time preconditions and postconditions, we chose not to invent more terms.

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 Figure 3b). 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`.

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 5, 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 [Per87-89a].

4.1 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 left-associative operator $\oplus$ denotes the postcondition propagation operator. Figure 4 depicts this testing of preconditions and the propagation of postconditions. 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.

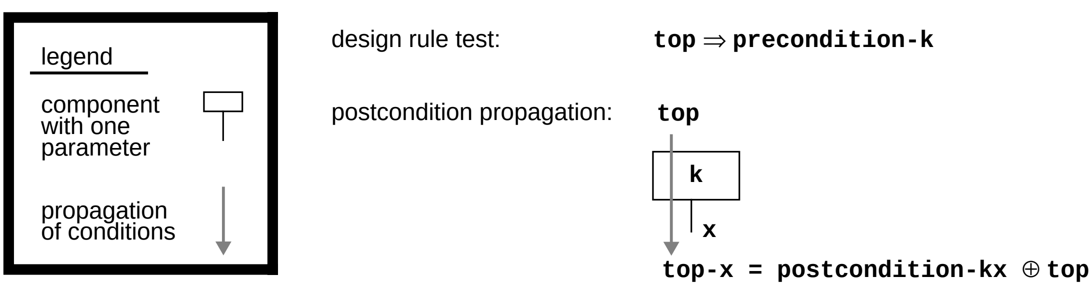

Figure 4: Top-Down Test and Propagation for Components with a Single Parameter

In general, type equations are trees of components, where branching arises when components have multiple parameters. Every parameter of a component has its own postcondition which defines the set of attribute values that hold for that parameter; these are the values that are propagated to any system instantiating that parameter. Postconditions for different parameters are generally not the same. As an example, the realm (type) of a parameter could be expressed as a postcondition. Thus, if a component had two parameters and the realms for both parameters were different, so too would be their postconditions.

Figure 5 depicts the general situation. Component **d[x,y]** has a postcondition for parameter **x** (**postcondition-dx**) and a postcondition for parameter **y** (**postcondition-dy**). If **top** is the set of conditions that hold prior to component **d** in a type equation, **top-x** is the set of conditions that hold for parameter **x** after **d** has been applied, and **top-y** is the set of conditions that hold for parameter **y**. **top-x** is computed by applying **x**'s postcondition to **top** (**top-x** = **postcondition-dx** ⊕ **top**) and **top-y** is computed similarly (**top-y** = **postcondition-dy** ⊕ **top**).

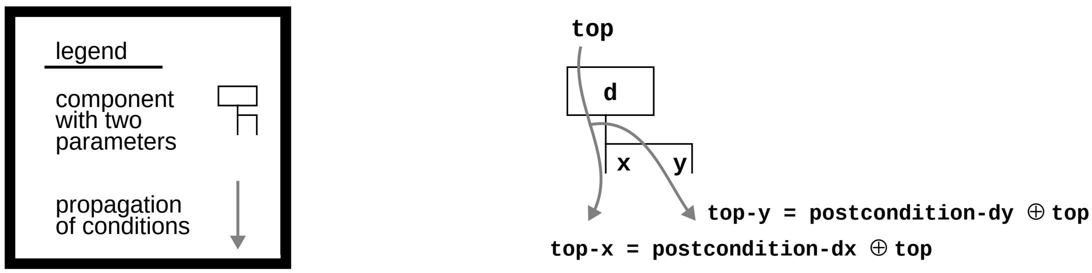

Figure 5: Top-Down Propagation for Components with Multiple Parameters

Given the operators ⊕ and ⟹, there is a straightforward, recursive algorithm for the top-down propagation of postconditions and the testing of component preconditions (see Appendix).

4.2 Bottom-Up Design Rule Checking

Every parameter of a component has preconditions (called *prerestrictions*) for instantiation; every component also has postconditions (called *postrestrictions*) that are exported to higher-level layers in a type equation. Figure 6 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.

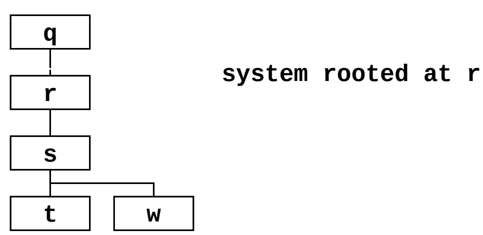

Figure 6: System Instantiation of Parameters

This gives rise to a different interpretation of instantiation, namely that *systems* instantiate parameters, not *components*. Every system exports a realm interface plus a set of attribute values (called *system postrestrictions*) that higher-level layers can reference. A component parameter has been correctly instantiated if the postrestrictions of the instantiating system imply that parameter's prerestrictions.

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**). Figure 7 depicts this testing of prerestrictions and the propagation of postrestrictions. Note that the same operators $\Longrightarrow$ and $\oplus$ used in top-down design rule checking are used in bottom-up design rule checking.

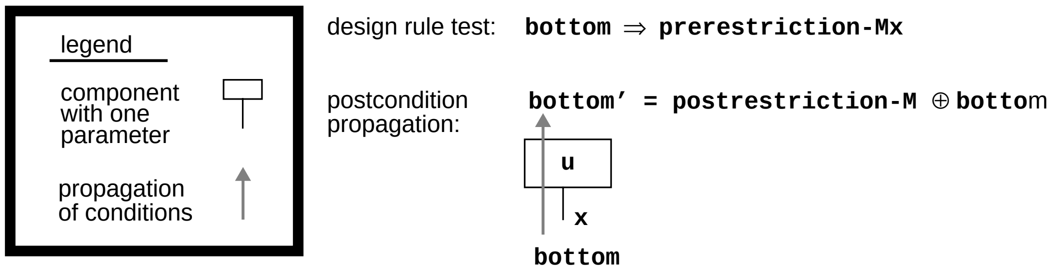

Figure 7: Bottom-Up Test and Propagation for Components with a Single Parameter

When components have multiple parameters, an additional design rule checking operator is needed. Recall component **`d[x,y]`**. Suppose system **`X`** instantiates parameter **`x`** and system **`Y`** instantiates **`y`**. The conditions that will be exported by the system **`d[X,Y]`** are determined by the postrestrictions of **`d`** applied to a merging of the conditions exported by systems **`X`** and **`Y`**. Δ is the merging operator. That is, the postrestrictions of the system **`d[X,Y]`** equals **`postrestriction-d`** $\oplus\,\Delta$**`(postrestriction-X, postrestriction-Y)`**. In theory, every component may have its own way of merging postrestrictions (i.e., properties of imported systems may be selectively propagated), thus the Δ operator may be component-specific. However, as we will see in Section 5.3, our experience suggests that domains rely only on a few Δ operator definitions.

Given the operators $\oplus$, $\Longrightarrow$, and Δ, there is a simple, recursive algorithm for the bottom-up propagation of postrestrictions and the testing of parameter prerestrictions (see Appendix).

4.3 Attribute Grammars

McAllester [McA94] observed that the concepts of realms, components, attributes, top-down and bottom-up design rule checking can be unified by attribute grammars [Aho88]. From previous sections, we know that realms of components define a grammar. Attributes model states of system (type equation) development, where 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 **`lex`** and **`yacc`**, are well-suited for specifying design rule checkers, as we'll see in Section 6.

5 Targeting DRC Algorithms for Specific Domains

The design rule checking algorithms of Section 4 are domain-independent. To specialize them to a particular domain, we need definitions and representations for attributes, predicates, and the operators $\oplus$, $\Longrightarrow$, and Δ. In the following, we explain the representations that we implemented for P2; virtually the same representations were used in Genesis.

5.1 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 [Per87-9a]) that in practice we can define them informally as attributes that assume restricted values. These values (**any**, **assert**, **negate**, **inherit**, and **set**) are defined in Table 1.[7]

Example P2 attributes are: **df_present** and **retrieval**. **df_present** represents the property that a component realizes logical deletions. That is, instead of physically deleting an element, 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 irrelevant to a component.

Attribute Value	Interpretation
any	nothing is known about the property of attribute
assert	property of attribute is asserted
negate	property of attribute is negated
inherit	property value is inherited from existing conditions
set	property of attribute is both asserted and negated

Table 1: Attribute Values used in P2 and Genesis

5.2 Predicates

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

Predicate	Interpretation
true (any)	true (no constraints)
assert	attribute has assert value
negate	attribute has negate value
set	attribute has the set value
not assert	attribute does not equal assert
not negate	attribute does not equal negate
not set	attribute does not equal set
false	false (unsatisfiable)

Table 2: Primitive Predicates used in P2 and Genesis

5.3 Postcondition Propagation Operator $\oplus$

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

7. **set** is a value that only arises during the merge of the postrestrictions of two or more systems, where one system **assert**s a property while another **negate**s this same property.

and the vector of existing conditions $E = [E_1, E_2, \ldots, E_n]$, the $\oplus$ operator is vector addition (using the + operator of Table 3):

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

Postcondition/Postrestriction + Existing Condition		Existing Condition: true	assert	set	negate	false
Postcondition or Postrestriction	assert	assert	assert	assert	assert	assert
	set	set	set	set	set	set
	negate	negate	negate	negate	negate	negate
	inherit	true	assert	set	negate	false

Table 3: The Propagation Operator + for a Single Attribute

5.4 Implication Operator $\Rightarrow$

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

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

Existing Condition → Precondition/ Prerestriction		Precondition or Prerestriction: true	assert	set	negate	not assert	not set	not negate	false
Existing Condition	true	true	false	false	false	true	true	true	false
	assert	true	true	false	false	false	true	true	false
	set	true	false	true	false	false	false	false	false
	negate	true	false	false	true	true	true	false	false
	false	true	true	true	true	true	true	true	true

Table 4: The Implication Operator → for a Single Attribute

5.5 The Merge Operator Δ

A characteristic of the P2 and Genesis domain models is that most components share the same Δ operator. In general, the "merge" of the postrestrictions of *n* systems corresponds to copying of the postrestrictions of the first system and discarding the postrestrictions of the rest. That is:

$$\Delta(\ \mathbf{postrestriction}_1,\ \mathbf{postrestriction}_2,\ \ldots\ ) = \mathbf{postrestriction}_1$$

The reasons for this are rather involved and peculiar to the domain of data structures and databases (see [Bat85] for further justification).

6 Implementation Notes

The implementation of our DRC algorithms and the P2/Genesis specializations of the ⊕, ⟹, and Δ operators was straightforward: the source files consist of 1500 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. Figure 8a 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 **`df_present`** property. Another design rule is to assert to layers above and below that **`array`** is a retrieval layer. Such a declaration is expressed by asserting the **`retrieval`** property as a postcondition and postrestriction.

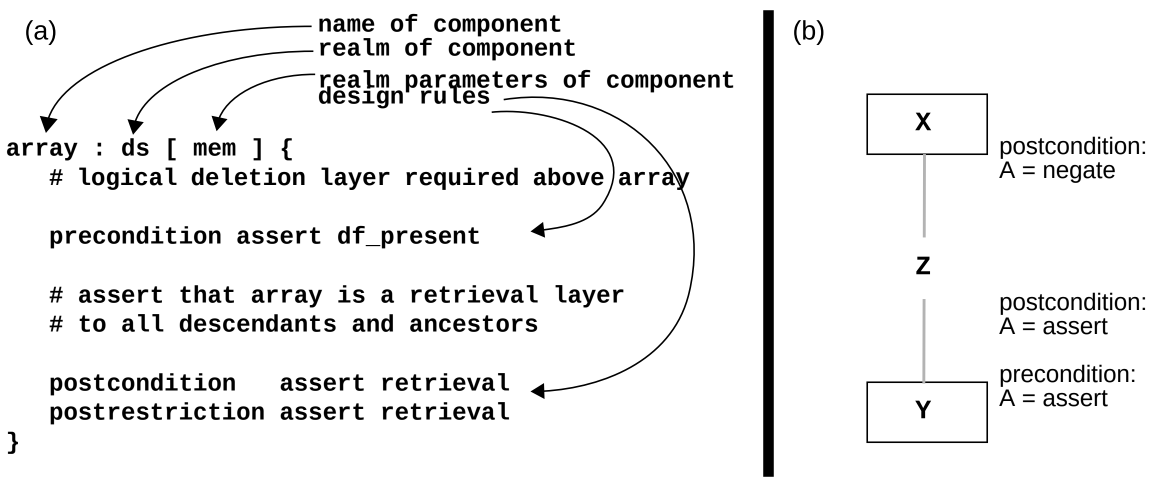

Figure 8: (a) Specification of Design Rules and (b) Locating and Fixing Errors

Algorithm Efficiency. Our DRC algorithms are efficient. Their complexity is $O(mn)$, where m is the number of attributes and n is the number of components in a type equation. To give readers upper estimates of m and n, the most complicated type equations that we have encountered in Genesis and P2 have approximately 30 components (i.e., $n \leq 30$). Genesis maintains the greatest number of attributes (m=14), whereas P2 has fewer (m=8), even though both generators maintain a library 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.

Explanation-Based Error Reporting. Detecting precondition and prerestriction errors is only part of the problem of debugging type equations; repairing equations are also important. One technique used in Inscape that we found particularly effective, called *precondition ceilings*, is illustrated in Figure 8b. 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 illustrates the idea.

Example. Suppose we would like a P2 container implementation that stores elements onto a binary tree, whose nodes are stored sequentially in transient memory. A first attempt at a composition might be:[8]

```
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) 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 (from Figure 8a) 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 ] ] ] ] ];
```

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.

Improvements. Design rule checking is rich area for research. There are many ways in which we could improve our model; four are discussed here.

First, it is possible to distinguish different levels of error severity by labeling predicates with their error strengths. Benign errors (such as the unnecessary redundancy of components) are reported to users by **`dreck`**, while fatal errors terminate code generation.

Second, the notation that we adopted in Section 2 does not indicate that components often have non-realm parameters. Such parameters, called *configuration parameters* [Cog93], include data types, tuning constants, performance constraints, etc. Configuration parameters are presently checked during the compilation of P2 programs or their corresponding C programs. We believe that a unified treatment of DRC for realm and configuration parameters is possible.

8. **`bintree`** links elements of a container onto a binary tree; the nodes of the binary tree will be stored sequentially in an **`array`**; the array will reside in **`transient`** memory. The **`top2ds`** layer must root all P2 type equations; had **`top2ds`** been absent, the DRC algorithms would report additional errors.

Third, although we have a general model of design rule checking, DRC software (algorithms, attributes, predicates, and DRC operators) must still be coded from scratch. We believe that a domain-independent tool can be created that eliminates the burden of DRC software development. Generalizing attributes types, predicates, and DRC operators — without sacrificing automatic DRC — is the key issue [Per89b].

Fourth, it may be possible to be more aggressive in repairing composition errors. For example, it seems likely that some errors can be repaired automatically (e.g., inserting **`inbetween`** into an equation). Also, it may be possible to identify the specific list of components (e.g., **`Z`** in Figure 8b) which could be used in repairing errors, thus further simplifying the task of debugging equations.

7 Related Work and Insights

Related Work. Shallow consistency checking is certainly not new to generators. DRACO, for example used a form of shallow consistency checking (called assertions and conditions) in composing layers of transformations [Nei80]. An early version of our DRC algorithms appeared in DaTE, the design rule checker for Genesis [Bat92b]. DaTE only supported component preconditions; there were no prerestrictions. 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 [McA94]. Preconditions and prerestrictions are treated uniformly as constraints. The constraints associated with a component are expressed as a VAG program. When an avionics system is composed from components, the set of constraints that must be satisfied is defined by the composition of corresponding VAG programs. The VAG interpreter has 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 [Gog83] and library interconnection languages, such as LIL and LILEANNA [Gog86, Tra93], are basic. The RESOLVE project explores the design of reusable and parameterized components, component certifiability, and the certifiability of component compositions [Sit94]. There are many similarities among these works and ours. One similarity is that GenVoca type equations are a simple module/library interconnection language. However, there is a basic difference: there is no "action-at-a-distance". 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 [Per92, Gar94-95, Mor94]. 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 basic 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 fifty components, and the number of primitive predicates that we have encountered in modeling different and multiple domains is modest (i.e., O(10)). 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 states 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 [Rum91, Boo91]. 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. Standardization substantially simplifies software composition (c.f., [Gar95]). Design rule checking is another example: standardization seems to limit the number of ways in which components can constrain each other's behavior. This, in turn, makes DRC tractable.

8 Conclusions

Software system generators are becoming important tools for software developers. These generators utilize libraries of reusable components to assemble complex, high-performance systems quickly and inexpensively. Each library component will have limitations, called design rules, on how it can be combined with other components. Experience has shown that validating component compositions is difficult to do by casual inspection; as the number of components and the complexity of their rules grow, a mechanical approach to validation is absolutely essential.

We have shown that a GenVoca domain model (or reference architecture model) is an attribute grammar, where sentences of the grammar define valid compositions of components. We have shown how the shallow consistency checking approach of Perry's Inscape environment can be adapted to exploit the semantics of GenVoca layers to define the actions of GenVoca production rules. Our approach distinguishes predicates and properties of component usage from those of parameter instantiation. We have shown (and our experience confirms) that domain-specific instances of our algorithms are practical: they are simple, easy to implement, and efficient. Moreover, they offer powerful explanation-based error reporting capabilities to suggest how incorrect compositions can be repaired.

Finally, we have observed that the number of attributes (primitive predicates) that need to be maintained for design rule checking GenVoca components is rather small. This is in contrast to small-scale components (i.e., functions) where the number of primitive predicates to be maintained can become large. We believe the explanation for this lies in the power of standardization 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.

So that others may learn from our work, **`dreck`** is 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's shallow consistency checking and his comments and insights on an earlier draft of this paper. We also thank Ira Baxter, Paul Clements, Dave Weiss, Chris Lengauer, Bruce Weide, and Steve Edwards for their clarifying comments.

9 References

[Aho88] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts, 1988.

[Bat85] D. S. Batory, "Modeling the Storage Architectures of Commercial Database Systems." *ACM Transactions on Database Systems*, December 1985.

[Bat92a] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM Transactions on Software Engineering and Methodology*, October 1992.

[Bat92b] D. S. Batory and J. R. Barnett. "DaTE: The Genesis DBMS Software Layout Editor." In *Conceptual Modeling, Databases, and CASE*, Pericles Loucopoulos and Roberto Zicari, eds. John Wiley & Sons, New York, New York. 1992.

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

[Bat94] D. Batory, J. Thomas, and M. Sirkin, "Re-engineering a Complex Application Using a Scalable Data Structure Compiler", *Proc. ACM SIGSOFT* 1994.

[Bat95] D. Batory and B.J. Geraci, "Validating Component Compositions in Software System Generators", Dept. Computer Sciences, TR-95-03, University of Texas at Austin, 1995.

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

[Bla91] L. Blaine and A. Goldberg, "DTRE - A Semi-Automatic Transformation System", in *Constructing Programs from Specifications*, Elsevier Science Publishers, 1991.

[Boo91] G. Booch. *Object-Oriented Design With Applications*, Benjamin-Cummings, 1991.

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

[Coh95] S. Cohen, R. Krut, S. Peterson, and J. Withey, "Models for Domains and Architectures: A Prescription for Systematic Software Reuse", *10th AIAA Computing in Aerospace*, 1995.

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

[Gar94] D. Garlan, R. Allen, and J. Ockerbloom, "Exploiting Style in Architectural Design Environments", *ACM SIGSOFT 1994.*

[Gar95] D. Garlan, R. Allen, J. Ockerbloom, "Architectural Mismatch or Why It's Hard to Build Systems out of Existing Parts", *Proc. ICSE 1995*, Seattle, 179-185.

[Gog83] J.A. Goguen, "Parameterized Programming", *Workshop on Reusability in Programming*. Newport, Rhode Island, September 1983.

[Gog86] J.A. Goguen, "Reusing and Interconnecting Software Components", Computer. February 1986, 16-28.

[Gom94] H. Gomaa, L. Kerschberg, V. Sugumaran, C. Bosch, and I. Tavakoi, "A Prototype Domain Modeling Environment for reusable Software Architectures", *IEEE International Conference on Software Reuse*, Rio de Janeiro, November 1994.

[Gra92] M. Graham and E. Mettala, "The Domain-Specific Software Architecture Program", *Proceedings of DARPA Software Technology Conference*, 1992. Also, in *Crosstalk: The Journal of Defense Software Engineering*, October 1992.

[Gri94] M.L. Griss and K.D. Wentzel, "Hybrid Domain-Specific Kits for a Flexible Software Factory", *Proceedings of SAC'94*, ACM, March 1994.

[Hei93] J. Heidemann and G. Popek, "File System Development with Stackable Layers", *ACM Transactions on Computer Systems*, March 1993.

[Hut91] N. Hutchinson and L. Peterson, "The x-kernel: An Architecture for Implementing Network Protocols", *IEEE Transactions on Software Engineering*, January 1991.

[Joh92] R.E. Johnson, "Documenting Frameworks using Patterns", *OOPSLA 1992*, 63-76.

[Lei94] J.C.S. do Prado Leite, M. Sant'Anna, and F.G. de Freitas, "Draco-PUC: A Technology Assembly for Domain-Oriented Software Development", *IEEE International Conference on Software Reuse*, Rio de Janeiro, November 1994.

[McA94] D. McAllester. "Variational Attribute Grammars for Computer Aided Design." ADAGE-MIT-94-01.

[Mor94] M. Moriconi and X. Qian, "Correctness and Composition of Software Architectures", *ACM SIGSOFT 1994*.

[Nei80] J. Neighbors, "Software Construction Using Components", Ph.D. Thesis, TR-160, ICS Department, University of California at Irvine, 1980.

[Nin94] J.Q. Ning, K. Miriyala, and W. Kozaczynski, "An Architecture-Driven, Business-Specific, and Component-Based Approach to Software Engineering", *IEEE International Conference on Software Reuse*, Rio de Janeiro, November 1994.

[Par76] D.L. Parnas, "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering*, March 1976.

[Per87] D.E. Perry, "Software Interconnection Models", *Proc. ICSE 1987*, 61-69.

[Per89a] D.E. Perry, "The Logic of Propagation in The Inscape Environment", *ACM SIGSOFT 1989*, 114-121.

[Per89b] D. E. Perry, "The Inscape Environment", *Proc. ICSE 1989*, 2-12.

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

[Rum91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[Sit94] M. Sitaraman and B. Weide, "Component-Based Software using RESOLVE", *ACM Software Engineering Notes*, October, 1994.

[Tra93] W. Tracz, "LILEANNA: A Parameterized Programming Language," *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability.* Lucca, Italy. R. Prieto-Dìaz and W.B. Frakes, eds. IEEE Computer Science Press, 1993.

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

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

10 Appendix

Notations for referencing a component, its children, and its design rules are listed in Table 5.

Notation	Meaning
k.parent	the parent component of **k**
k.child$_i$	the component that instantiates parameter *i* of **k**
k.pre	the precondition of **k**
k.post$_i$	the postcondition of parameter *i* of **k**
k.preres$_i$	the prerestriction of parameter *i* of **k**
k.postres	the postrestriction of **k**
k.cpost	the cumulative postcondition of **k**'s ancestors
k.cpostres	the cumulative postrestriction of the system rooted at **k**
k.Δ	the postrestriction merge operator of **k**

Table 5: The structure of a component **k**

Our algorithms for top-down and bottom-up design rule checking are listed in Figure 8 and Figure 9 respectively. A type equation is DRC correct if there are no precondition and prerestriction validation errors.

```
// if te is the root component of a type equation and
// top defines the initial attribute states for an abstract program,
// precondition_validation( te, top ) will return TRUE
// if te is free of precondition errors

boolean precondition_validation( root, root_conditions )
{
   root.cpost = root_conditions;
   if (root.cpost ⇒ root.pre) {
      foreach child i of root {
         cpost = root.post_i ⊕ root.cpost;
         if (¬ precondition_validation( root.child_i, cpost ))
            return FALSE ;
      }
      return TRUE ;
   }
   return FALSE;
}
```

Figure 9: Precondition Validation Algorithm

```
// if te is the root component of a type equation and
// goal is the prerestriction that te is to satisfy,
// prerestriction_validation(te,goal) returns TRUE if
// te has no prerestriction errors and that it satisfies goal

boolean prerestriction_validation( root, root_prerestriction )
{
   if (root has no children) {
      root.cpostres = root.postres;
   } else {
      foreach child i of root {
         // return false if any subtree has prerestriction errors

         if ( ¬ ( prerestriction_validation(root.child_i, root.preres_i ) )
            return FALSE;
      }
      root.cpostres = root.postres ⊕
            root.Δ( root.child_1.cpostres, root.child_2.cpostres, ...) ;
   }
   return root.cpostres ⇒ root_prerestriction ;
}
```

Figure 10: Prerestriction Validation Algorithm