

## Design Rule Checking

Not all syntactically correct combinations of GenVoca components are semantically correct. Some components work only in the presence (or absence) of other components.

Fundamental problem: impossible for generator users to debug generated code; need automated help to debug component compositions.

Design rules are domain-specific constraints that specify illegal configurations of components. *Design rule checking (DRC)* is the process of (automatically) applying design rules.

In this lecture, we present:

- a model of DRC based on attribute grammars
- instance of model that works for P3 and Genesis
- relate DRC to research on software architectures

## Motivating Example: P3

Generator for container data structures

- relies on two realms containing 50 components:

```
ds = { bintree[ ds ],      // binary tree
       dlist[ ds ],       // unordered list
       odlist[ ds ],      // ordered list
       avail[ ds ],       // free list manager
       array[ mem ],      // sequential storage
       malloc[ mem ],     // random storage
       ... }

mem = { transient,        // in memory storage
       persistent,       // memory mapped
       }
```

Data structures are modeled by type equations

- reference 5 to 15 components
- too elaborate to validate by inspection
- some components have obscure rules for their use

## Example P3 Design Rule

`inbetween` component encapsulates:

- algorithms shared by many data structure components (e.g., `bintree` and `dlist`)
- deals with positioning of cursor after element is deleted
- details complex, hidden from user

Correct usage of `inbetween` requires:

- one copy in TE that has 1+ data structure components &
- precede all such components in equation

```
right = ...inbetween[ ... [dlist[ bintree[] ]]];
wrong = ... dlist[ inbetween[ bintree[] ]];
```

Such rules should not be borne by programmers

- too easy to forget and be misapplied

***want rules tested automatically***

## Results from Software Architectures

Perry's *Inscope* (1989) is environment for managing evolution of software systems:

- novel aspects: obligations and consistency checking

Components have pre-, post-conditions, and obligations

*bank loan example*

- *obligations* are conditions that must be satisfied by system that uses the component
- require “action-at-a-distance” — nonlocally satisfied
- propagated to enclosing modules where they are eventually satisfied by some postcondition

*Full-fledged verification not attempted*

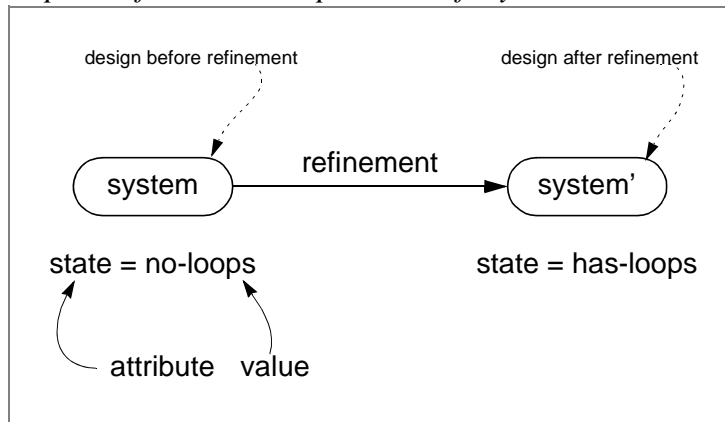
- primitive predicates declared (but informally defined)
- pre-, post-conditions, obligations expressed in terms of primitives
- practical & powerful form of “shallow” consistency checking using pattern matching and simple deductions

## Design Rule Checking

Adapt and generalize Inscape consistency checking to DRC by exploiting the semantics of GenVoca layers

(1) DRC models states of system (TE) *design*

- *not states of system execution*
- model states / properties of system design by assigning values to attributes
- *exploit refinement interpretation of layers*



## DRC Basics

(2) Preconditions and obligations of component K are satisfied “at-a-distance” by components that lie either:

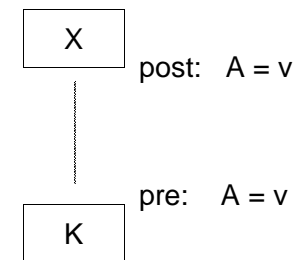
- (far) beneath K or
- (far) above K

*constraints typically not satisfied by adjacent components (c.f. Goguen, Tracz, Sitaraman references)*

Properties exported to “higher” layers generally not the same as properties exported to “lower” layers

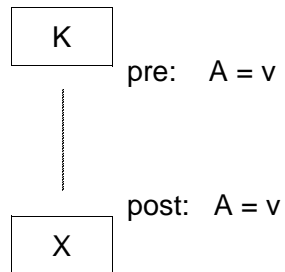
Leads to 2 kinds of design rules:

- #1: *preconditions for component usage*



## DRC Basics (Continued)

- #2: *preconditions for parameter instantiation*



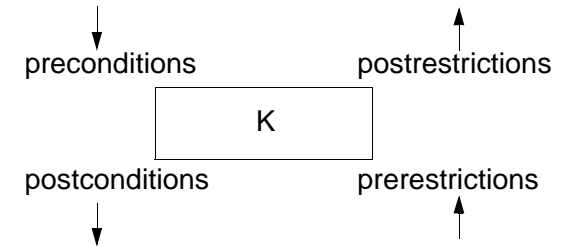
new names: preconditions called *prerestrictions*

postconditions called *postrestrictions*

note: prerestrictions correspond to Inscape obligations

## DRC Basics (Continued)

Components have:



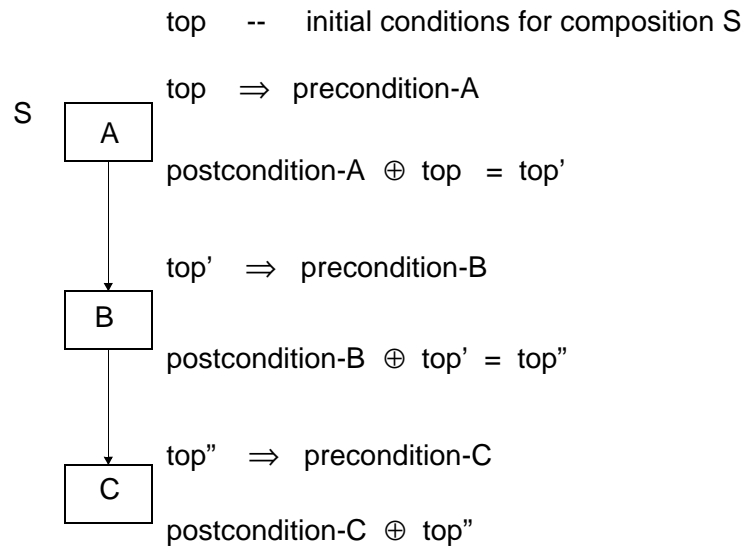
Design rule checking involves:

- top-down propagation of postconditions and testing component preconditions
- bottom-up propagation of postrestrictions and testing of parameter prerestrictions

In following, we assume no restriction on complexity of predicates, but will later show that very simple predicates suffice for P3 and Genesis.

## Top-Down DRC

Components have preconditions and postconditions

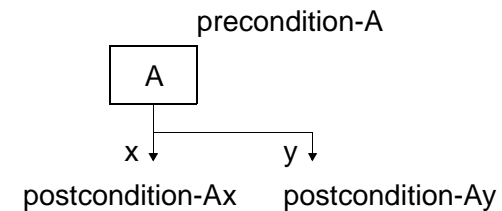


- postconditions propagated by  $\oplus$  operator
- conditions tested by  $\Rightarrow$  operator

## A Twist...

Consider component with multiple parameters: A[x,y]

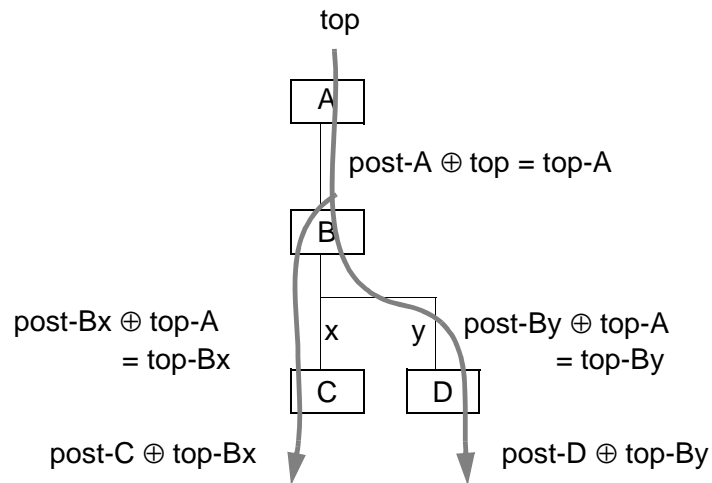
- gives rise to trees (dags)
- twist: each parameter has its own postcondition



i.e., conditions for parameter x may be different for those of y in A[x,y]

- example: the realm of a parameter could be expressed as a postcondition; realms for x and y could be different

## Top-Down Algorithm

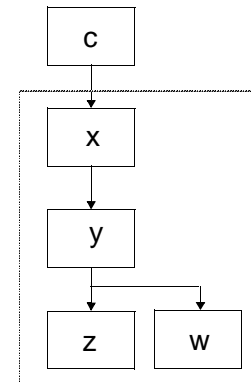


*simple recursive algorithm for top-down propagation of conditions and testing component preconditions*

## Bottom-Up DRC

Conditions must also be propagated upwards...

- parameters of components may have preconditions (called *prerestrictions*) for instantiations to be correct

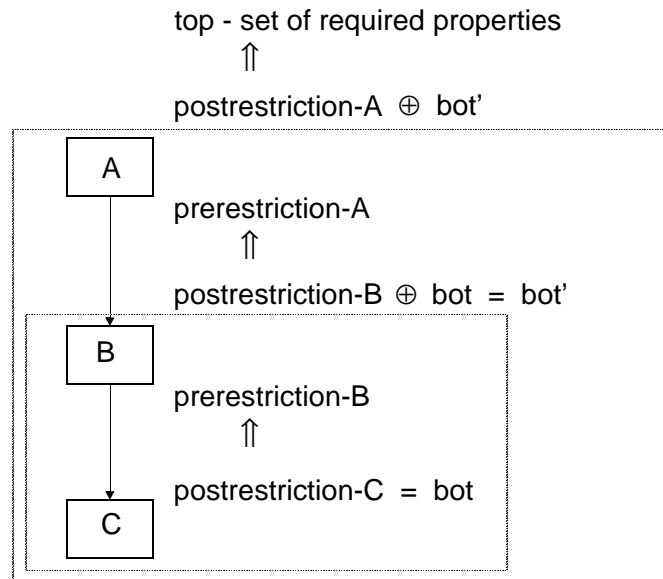


*prerestrictions for c are generally not satisfied by the component x that instantiates its parameter, but rather by components deep within the system rooted by x*

- systems instantiate parameters, not components
- exported states (called *postrestrictions*) propagated upwards so that prerestrictions can be tested

## Bottom-Up DRC Continued

Every component has *postrestrictions*, i.e., exported states, and *prerestrictions* for each parameter



- use same operators  $\oplus$  and  $\Rightarrow$  for bottom-up DRC
- simple recursive algorithm for bottom-up DRC

## Attribute Grammars

McAllester observed *attribute grammars* unify realms, attributes, top-down & bottom-up DRC algorithms

- realms of components modeled by grammars
- attributes model program development states
- postconditions are *inherited attributes* (values determined by ancestors)
- postrestrictions are *synthesized attributes* (values determined by descendants)

*Bonus: common tools (lex, yacc) well-suited for implementing design rule checkers*

Need to supply definitions & representations for:

- attributes
- predicates
- operators  $\oplus$  and  $\Rightarrow$

For the P3 and Genesis generators...

## P3 Attributes

Each attribute models a property that exposes a composition constraint

Attributes have restricted values:

Value	Interpretation
any	nothing is known about property
assert	property is asserted
negate	property is negated
inherit	attribute value inherited, but is otherwise unconstrained

- example:

attribute:            component\_belongs\_to\_realm\_A

attribute value:    assert (or negate)

- see Batory and Geraci *IEEE TSE 1997* paper (and report UTCS TR-94-03 for other values...)

## P3 Predicates

Preconditions & prerestrictions request specific attribute values (any, assert, negate), but not how they were determined (i.e., inherit)

- only 4 different *primitive* predicates:

Predicate	Interpretation
P-any	true (no constraints)
P-assert	attribute has <b>assert</b> value
P-negate	attribute has <b>negate</b> value
P-false	false (unsatisfiable)

- complex predicates are typically conjunctions of primitives, one primitive predicate for each attribute
- encode as vector of predicates indexed by attribute

$$P \equiv P_1 \wedge P_2 \wedge \dots \equiv [P_1, P_2, \dots]$$



## Postcondition Propagation Operator $\oplus$

Component postconditions assert or negate values, or may propagate values

- table below defines the condition propagation operator  $\oplus$  for a single attribute:

component postcondition + existing condition		component postcondition		
		inherit	assert	negate
existing condition	any	any	assert	negate
	assert	assert	assert	negate
	negate	negate	assert	negate

- given  $P = [P_1, P_2, \dots]$  and  $E = [E_1, E_2, \dots]$

$$P \oplus E = [P_1 + E_1, P_2 + E_2, \dots]$$

## Implication Operator $\Rightarrow$

The implication operator  $\rightarrow$  for a single attribute is:

Existing Condition $\rightarrow$ Precondition		Precondition			
		P-any	P-assert	P-negate	P-false
Existing Condition	assert	true	true	false	false
	negate	true	false	true	false
	any	true	false	false	false

- given  $E = [E_1, E_2, \dots]$  and  $P = [P_1, P_2, \dots]$

$$E \Rightarrow P = (E_1 \rightarrow P_1) \wedge (E_2 \rightarrow P_2) \wedge \dots$$

## Implementation Notes

Straightforward implementation: 1500 lines in `lex` & `yacc`

DRC algorithm is efficient:  $O(mn)$

$m$  = # of attributes,  $n$  = # of components

Example domain models:

Generator (Domain)	# of Realms	# of Components	# of Attributes
Genesis (databases)	9	52	14
P3 (data structures)	3	50	7

Some P3 attributes:

attribute	property description
logical_deletion	“a logical deletion layer”
retrieval	“a retrieval layer”

## Straightforward Specifications

Example component & design rule declaration:

```

array : ds [ mem ] {
    # logical del. layer required above array
    precondition    assert    logical_deletion

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

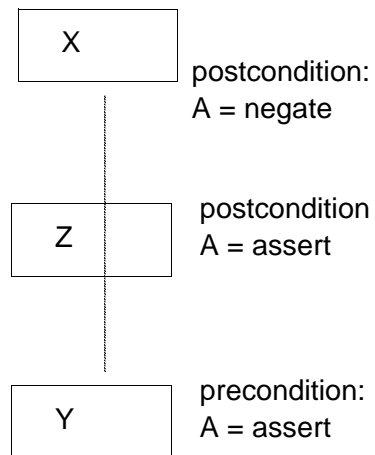
    postcondition    assert    retrieval
    postrestriction  assert    retrieval
}

```

## Explanation Based Error Reporting

In addition to detecting errors, we can extend DRC algorithms to suggest how to repair a type equation

- *precondition ceilings* from Inscape:



- error located in between components X and Y
- similar technique for obligations/prerestrictions

## Example

Example: want container implementation that stores elements onto a binary tree, whose nodes are stored sequentially in transient memory

First attempt at composition:

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

DRC response:

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
```

## Explanation Based Error Reporting

Clumsy fix...

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

DRC response:

Precondition error:

```
a retrieval layer (bintree) not expected above
    qualify
```

Correct type equation — swap `qualify` with `bintree`:

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

*DRC directs users to modify TE to the  
“nearest” correct TEs in space of all TEs*

## Insights

Why isn't DRC a challenging problem in program verification?

- solution unlikely to be automatable

Inscape work and our work have observed:

- problem is straightforward
- solution automatable and efficient

... why? ... 2 reasons

Reason #1:

- shallow consistency checking goes a very long way
- in general, most logical errors are shallow errors

conjecture: all errors at component composition level should be shallow

- remaining errors must be dealt with by component implementors

## Insights (Continued)

Reason #2: important distinction:

- Inscape components are functions
- GenVoca components are subsystems

Large software systems consist of tens of thousands of lines of code

- hundreds or thousands of functions

⇒ hundreds or thousands of primitive predicates

- TEs rarely have more than 50 components

⇒ modest # of primitive predicates in a domain ~10-40

seems counterintuitive

Why?

- modeling states of development (not execution) reduces number of properties to examine
- and GenVoca is a methodology for designing reusable components ...

## The Key

What makes OO designs so powerful and attractive?

- Ans: ability to manage and control software complexity

Standardization is powerful way of managing and controlling software complexity in a family of systems

*Standardization makes some problems tractable that would otherwise be very difficult*

- ex: composing off-the-shelf components
- composition of components is simple in GenVoca
- standardization seems to limit the ways in which components can constrain each other's behavior  
⇒ make DRC tractable

Historical perspective...

## Additional Insights

Recall **Example** lecture: a GenVoca domain model is implementation independent

- enhances power of design rule checking
- DRC tells you whether two refinements (concepts) can be composed *regardless how they are implemented*

ex: `bintree[ encrypt[...] ]` may be correct\*\*  
 ex: `encrypt[ bintree[...] ]` is always incorrect

exposes fundamental composition constraints of a domain

\*\* additional design rules may be added because of implementation constraints

- compositional imposes quite a few additional constraints
- transformational the least...

## Conclusions — Lessons Learned

- *GenVoca domain models (realms + design rules) are attribute grammars*

can use existing tools (lex, yacc) to express models

- *simple, efficient algorithms for DRC*

constraints imposed on higher layers (preconditions)

constraints imposed on lower layers (prerestrictions)

don't need formal methods, theorem provers

- *components that are designed to be interoperable, plug-compatible, and interchangeable often makes complex problems much easier to solve*

standardization of programming abstractions is a powerful way of controlling the complexity of a family of software systems

## Further Reading

- D. Batory and J. Barnett, "DaTE: The Genesis DBMS Software Layout Editor", in *Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development*, P. Loucopoulos and R. Zicari, editors, Wiley, 1992.
- D. Batory and B.J. Geraci, "Validating Component Compositions and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering*, February 1997, 67-82.
- L. Blaine and A. Goldberg, "DTRE - A Semi-Automatic Transformation System", in *Constructing Programs from Specifications*, Elsevier Science Publishers, 1991.
- J.A. Goguen, "Reusing and Interconnecting Software Components", *IEEE Computer*, February 1986.
- J. Neighbors, "Draco: A Method for Engineering Reusable Software Components", in T.J. Biggerstaff and A. Perlis, eds., *Software Reusability*, Addison-Wesley/ACM Press, 1989.
- M.D. Katz and D.J. Volper, "Constraint Propagation in Software Libraries of Transformation Systems", *Int. Journal Software Engineering and Knowledge Engineering*, Vol. 2 #3 (1992), 355-374.
- D. McAllester. "Variational Attribute Grammars for Computer Aided Design." ADAGE-MIT-94-01.
- M. Moriconi and X. Qian, "Correctness and Composition of Software Architectures", *ACM SIGSOFT 1994*.
- D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture", *ACM SIGSOFT Software Engineering Notes*, October 1992, 40-52.
- D.E. Perry, "The Inscape Environment", *Proc. ICSE 1989*, 2-12.

## Further Reading (Continued)

- D.E. Perry, "Software Interconnection Models", *Proc. ICSE 1989*, 61-69.
- D.E. Perry, "The Logic of Propagation in The Inscape Environment", *ACM SIGSOFT 1989*, 114-121.
- M. Sitaraman and B. Weide, "Component-Based Software Using RESOLVE", *ACM Software Engineering Notes*, October 1994.
- 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-Diaz and W.B. Frakes, eds. IEEE Computer Science Press, March 24-26, 1993.