Formalizing Design Patterns:
A Comprehensive Contract for Composite

Jason O. Hallstrom
School of Computing, CS Division
Clemson University

Neelam Soundarajan
Computer Science and Engineering
Ohio State University

This work is supported in part by the National Science Foundation
(CNS-0745846, DUE-0633506)

Responsibilities and Rewards

When using a pattern in an given application, designers are interested in two sets of properties

- Responsibilities
  The implementation requirements that must be satisfied to apply the pattern correctly
- Rewards
  The system properties that result by virtue of satisfying the implementation requirements

A comprehensive pattern formalism must capture both

The Formalization Challenge

The main challenge in formalizing patterns is striking the right balance between two competing objectives

- ❖ Precision
  Implementation requirements and behavioral guarantees must be clear and unambiguous
- ❖ Flexibility
  Pattern specifications must be customizable as appropriate to particular applications

A comprehensive pattern formalism must satisfy both

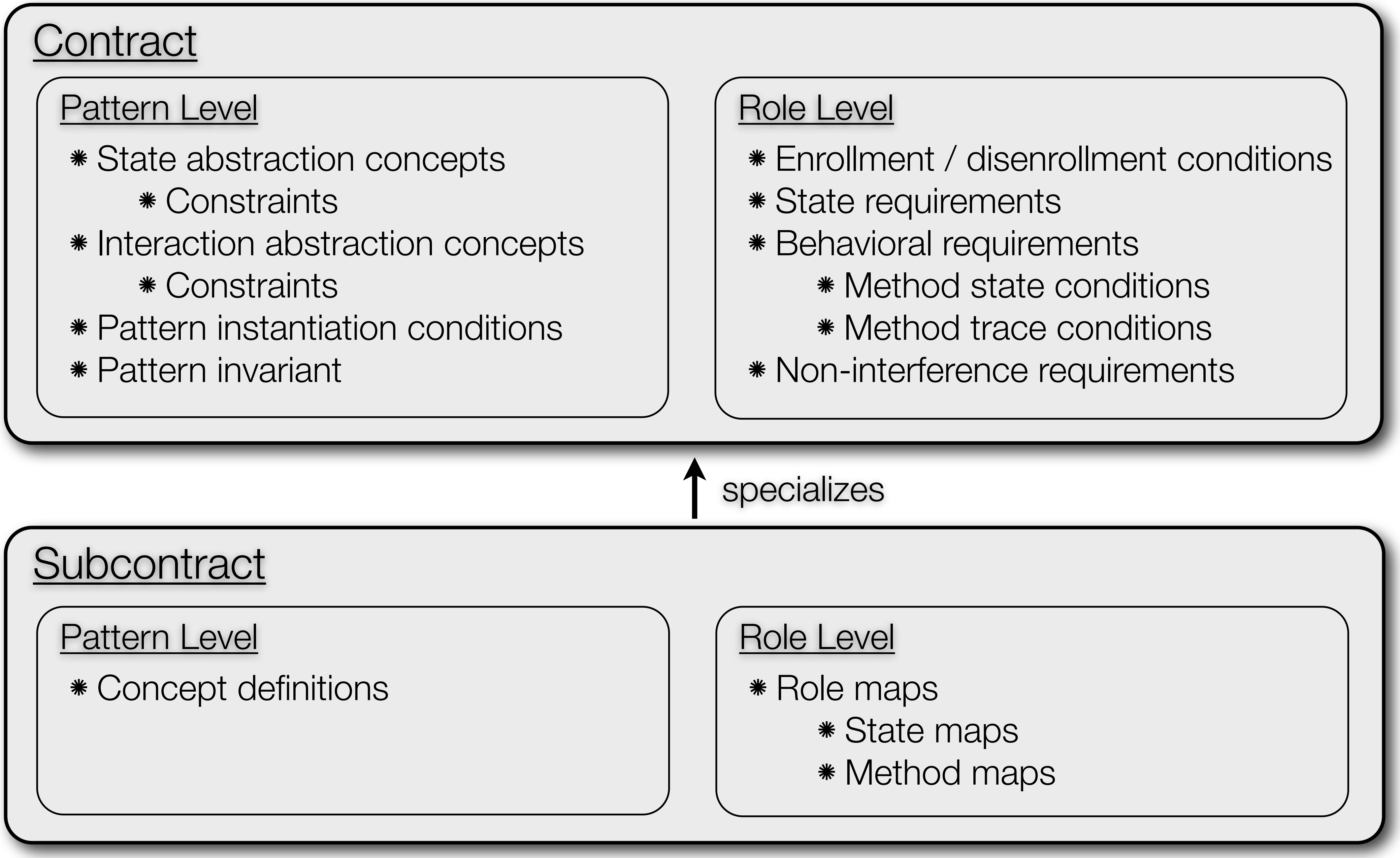
Pattern Contracts

Our approach to addressing these requirements relies on a multi-level contract framework

- Pattern Contract
  Captures the requirements and guarantees associated with *all* instances of a given pattern
- Pattern Subcontract
  Refines a pattern contract (or subcontract) to yield the specification of a sub-pattern or pattern implementation

Abstraction concepts are a key source of contract flexibility

Contract Structure



Contract

Pattern Level

- State abstraction concepts
    - Constraints
- Interaction abstraction concepts
    - Constraints
- Pattern instantiation conditions
- Pattern invariant

Role Level

- Enrollment / disenrollment conditions
- State requirements
- Behavioral requirements
    - Method state conditions
    - Method trace conditions
- Non-interference requirements

specializes

Subcontract

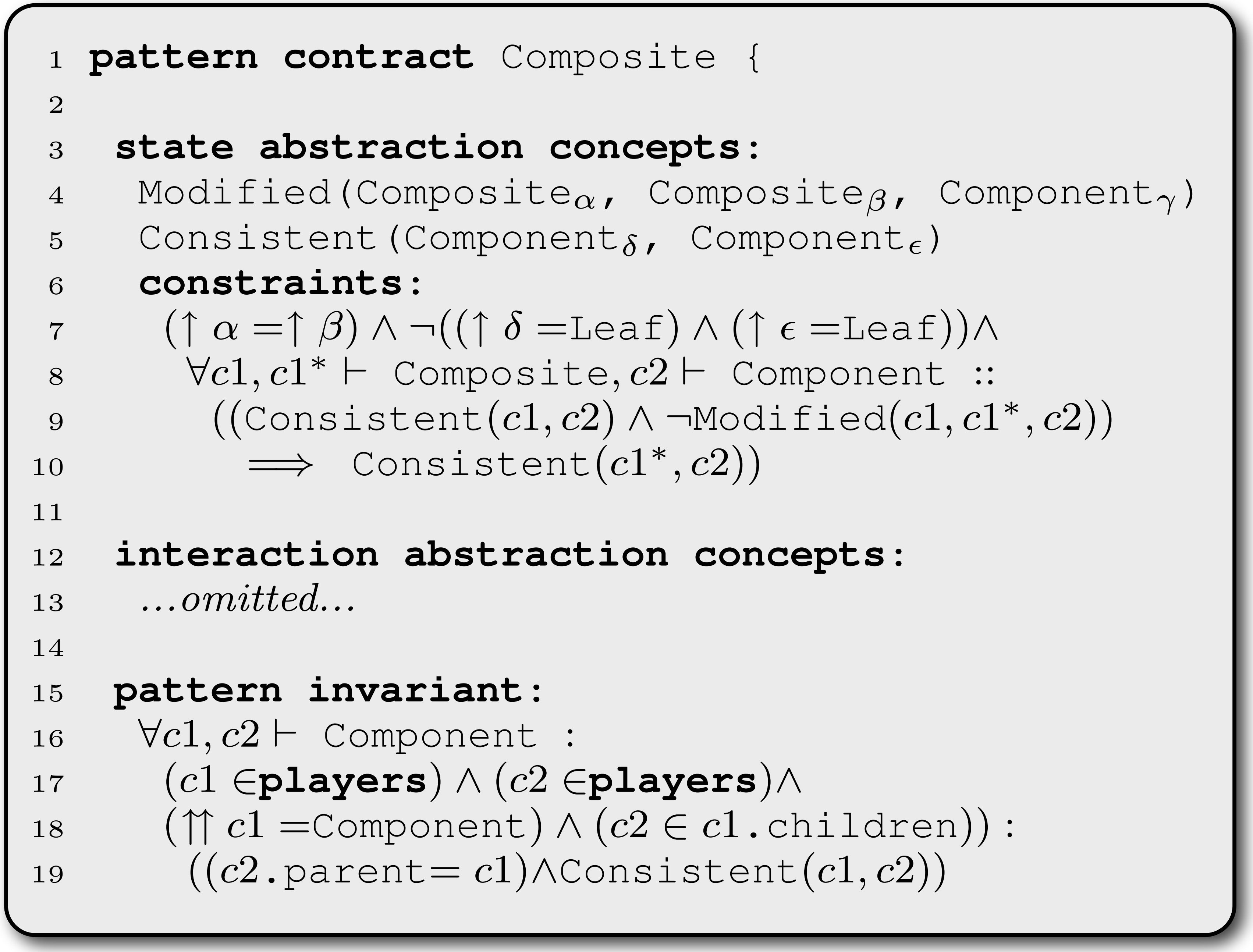
Pattern Level

- Concept definitions

Role Level

- Role maps
    - State maps
    - Method maps

Example: Composite Pattern (1/3)



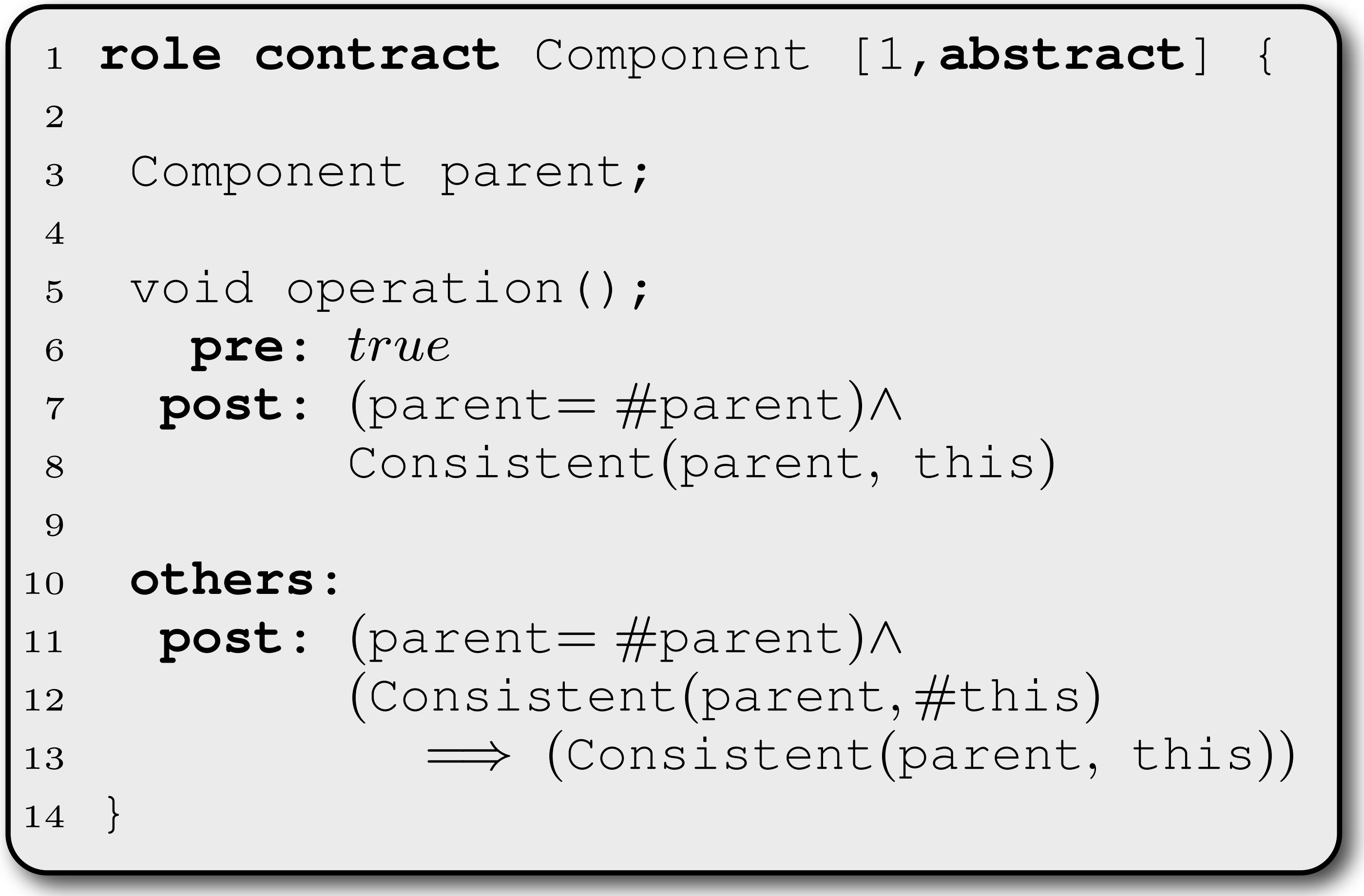
```
pattern contract Composite {

 state abstraction concepts:
  Modified(Composite$_\alpha$, Composite$_\beta$, Component$_\gamma$)
  Consistent(Component$_\delta$, Component$_\epsilon$)
  constraints:
   $(\uparrow \alpha =\uparrow \beta) \land \neg((\uparrow \delta =$Leaf$) \land (\uparrow \epsilon =$Leaf$))\land$
    $\forall c1, c1^* \vdash$ Composite, $c2 \vdash$ Component ::
     $(($Consistent$(c1, c2) \land \neg$Modified$(c1, c1^*, c2))$
      $\implies$ Consistent$(c1^*, c2))$

 interaction abstraction concepts:
  $\ldots$omitted$\ldots$

 pattern invariant:
  $\forall c1, c2 \vdash$ Component :
   $(c1 \in$players$) \land (c2 \in$players$)\land$
   $(\uparrow\uparrow c1 =$Component$) \land (c2 \in c1$.children$))$ :
    $((c2$.parent$= c1)\land$Consistent$(c1, c2))$
```

Example: Composite Pattern (2/3)

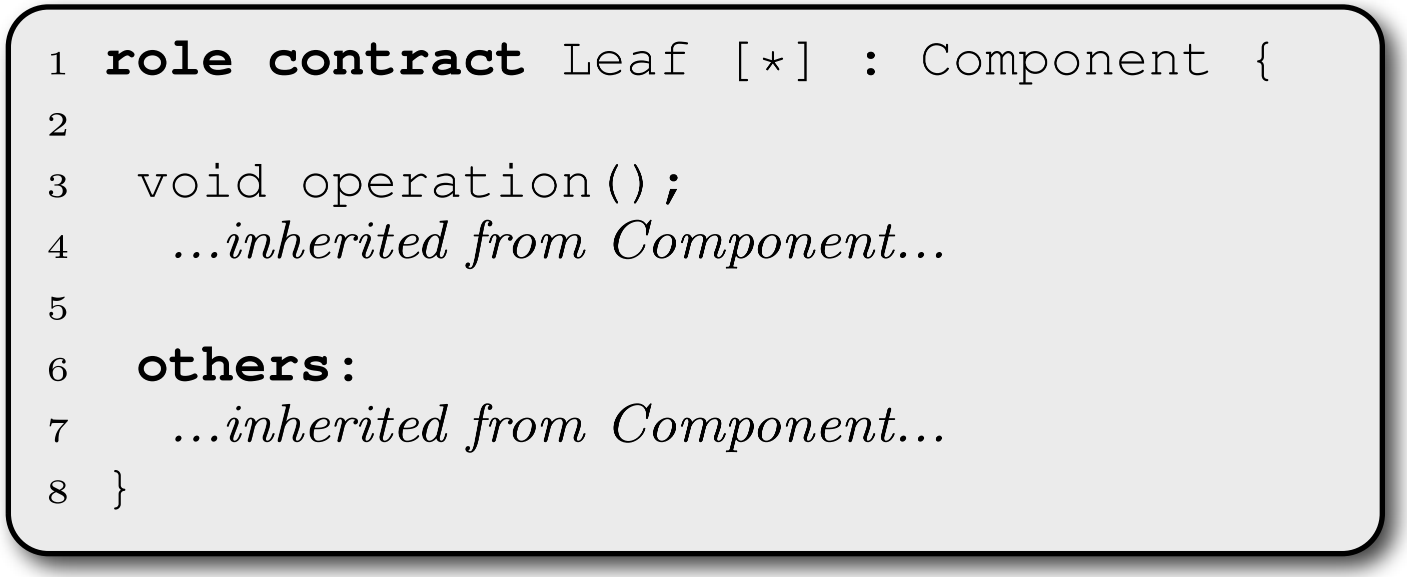


```
role contract Component [1,abstract] {

 Component parent;

 void operation();
   pre: true
  post: (parent= #parent)∧
        Consistent(parent, this)

 others:
  post: (parent= #parent)∧
        (Consistent(parent,#this)
           ⟹ (Consistent(parent, this))
}
```



```
role contract Leaf [⋆] : Component {

 void operation();
  ...inherited from Component...

 others:
  ...inherited from Component...
}
```

Example: Composite Pattern (3/3)

```
role contract Composite [+] : Component {

 Set<Component> children;

 void add(Component c);
   pre: c ∉ children
  post: (children= (#children∪{c}))∧
        (c.parent=this)∧
         ∀oc ⊢ Component :
          (oc ∈ #children):
           ¬Modified(this, #this, oc)∧
        (|τ.c.operation| = 1)

 void remove(Component c);
   pre: c ∈ children
  post: (children= (#children−{c}))∧
        ∀oc ⊢ Component :
         (oc ∈ #children):
          ¬Modified(this, #this, oc)

 ...other child management methods omited...

```

```
 void operation();
   pre: ...inherited from Component...
  post: ...inherited from Component...∧
        (children= #children)∧
        ∀c ⊢ Component :
         (c ∈children):
          (Modified(this, #this, c)
           ⟹ (|τ.c.operation| = 1))

 others:
   ...inherited from Component...∧
   (children=children)∧
   ∀c ⊢ Component :
    (c ∈ #children):
     ¬Modified(this, #this, c)
}
```

Questions?

Formalizing Design Patterns:

A Comprehensive Contract for Composite

Jason O. Hallstrom
School of Computing, CS Division
Clemson University

Neelam Soundarajan
Computer Science and Engineering
Ohio State University

This work is supported in part by the National Science Foundation
(CNS-0745846, DUE-0633506)