# Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure

Marwan Abi-Antoun
Dept. of Computer Science*
Wayne State University
mabiantoun@wayne.edu

* This work was conducted while a Ph.D. student at Carnegie Mellon University, under the supervision of Professor Jonathan Aldrich

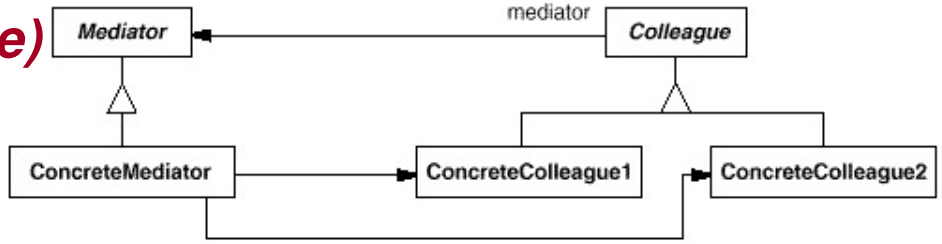
# **Software architecture: high-level description of a system's organization**

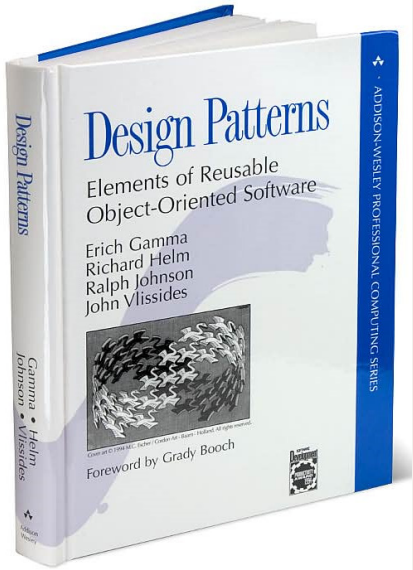
[Perry and Wolf, 1992] [Garlan and Shaw, 1993—] [Medvidovic *et al.*, 1995—]

- Communication between stakeholders
- **Qualitative** architectural evaluation
- **Quantitative** architectural analyses
- Different perspectives or **views**:
  - **Distinct** but **complementary**
  - Here, we focus on **structure** not **behavior**

# Structure

## *Class Diagram (Type structure)*



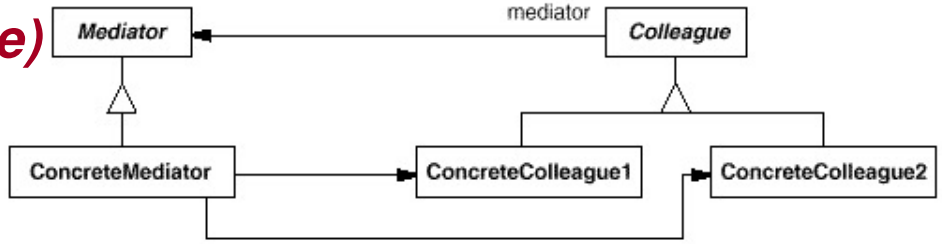



# Code architecture shows code structure (e.g., UML class diagram)

- **Static code structure** of system:
  - Classes, packages, modules, layers, ...
  - **Inherits from** class, **implements** interface
  - Dependencies: **imports**, call graphs, etc.
- Impacts qualities like **maintainability**
- **Mature** tool support

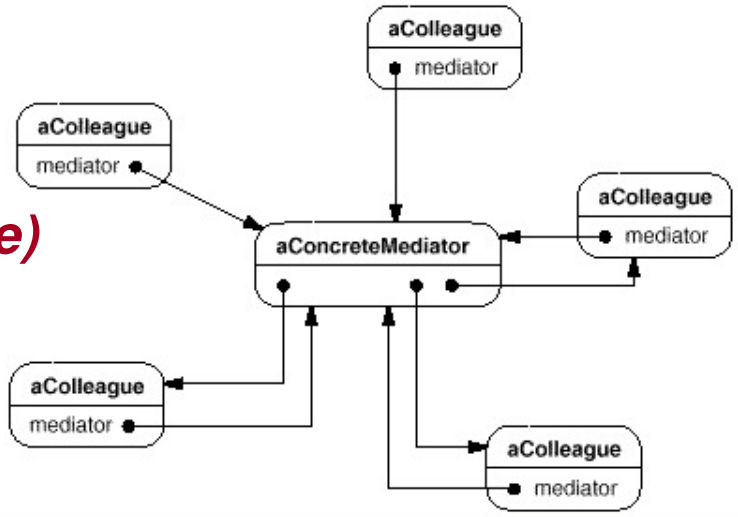
# Structure

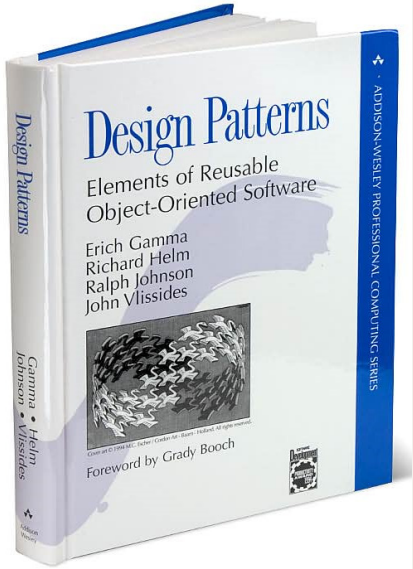
***Class Diagram (Type structure)***




A typical object structure might look like this:

***Object diagram (Instance structure)***



# **Runtime architecture** shows objects (e.g., **object diagram**) and their relations

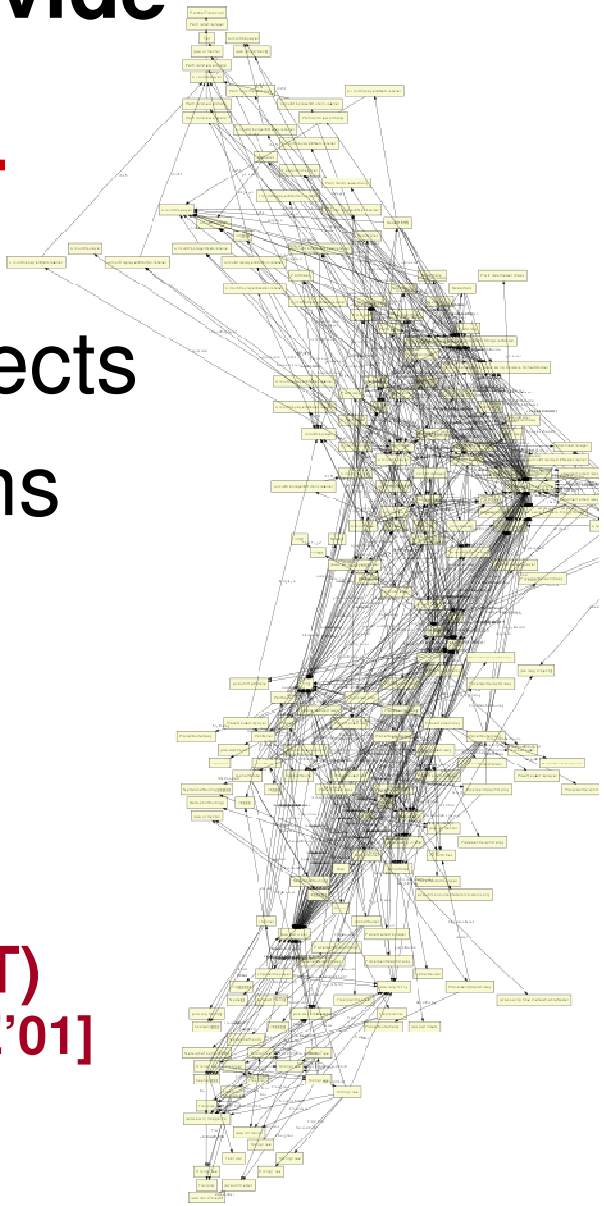
- **Runtime architecture** of system:
  - Runtime component = sets of **objects**
  - Runtime interaction = e.g., points-to relation
- Impacts qualities such as **security**, **performance**, **reliability**, etc.
- **Immature** tool support

# Architectural extraction: state-of-the-art

- Use **dynamic** analysis
  - Analyze one or more program runs
  - May **omit important objects** or **relations** that arise only **in other** program runs
- Use **static** analysis
  - Can capture **all possible** program runs
  - Extract low-level **non-architectural** views
  - Precise analyses often **do not scale**
  - Still an open problem

# Flat object graphs do not provide architectural abstraction

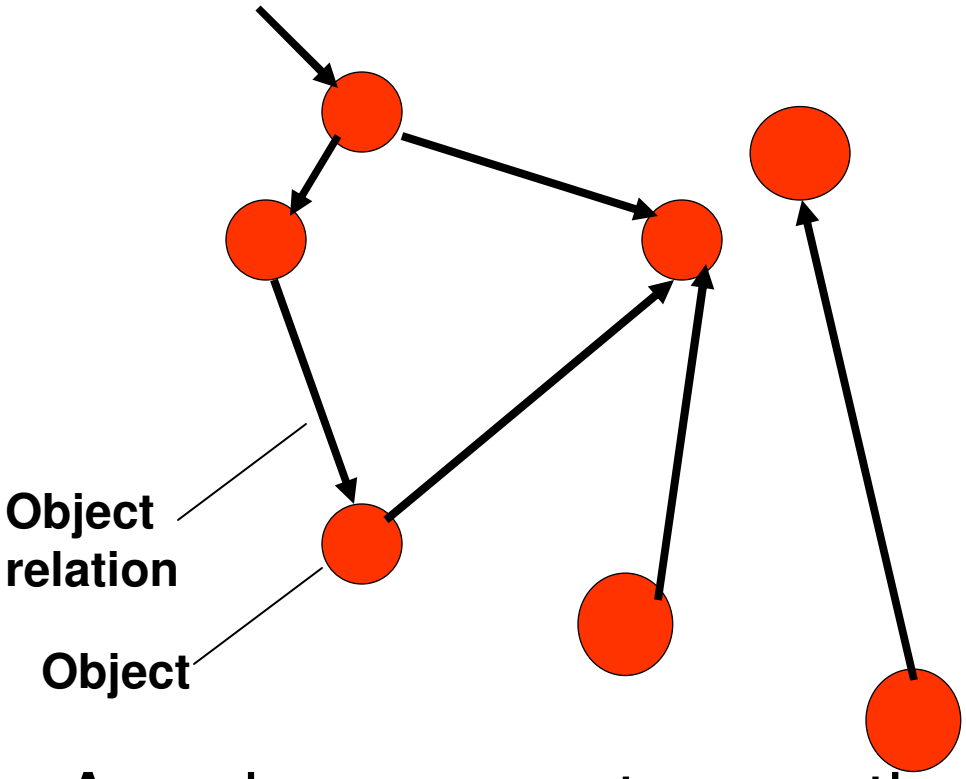
- Low-level objects mixed with architecturally significant objects
- No scale-up to large programs



**Output of WOMBLE (MIT)**
**[Jackson and Waingold, TSE'01]**
**on 8,000-line system.**

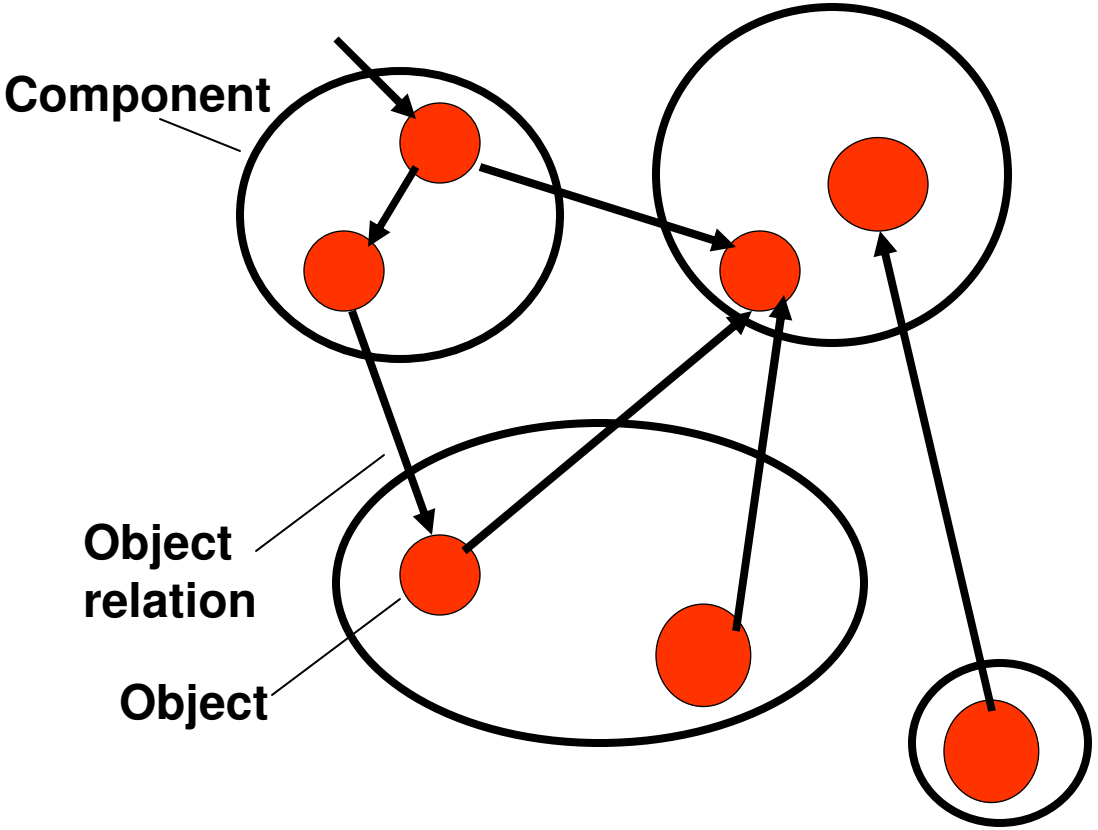
# Architectural abstraction

# At runtime, an object-oriented system appears as a Runtime Object Graph (ROG)

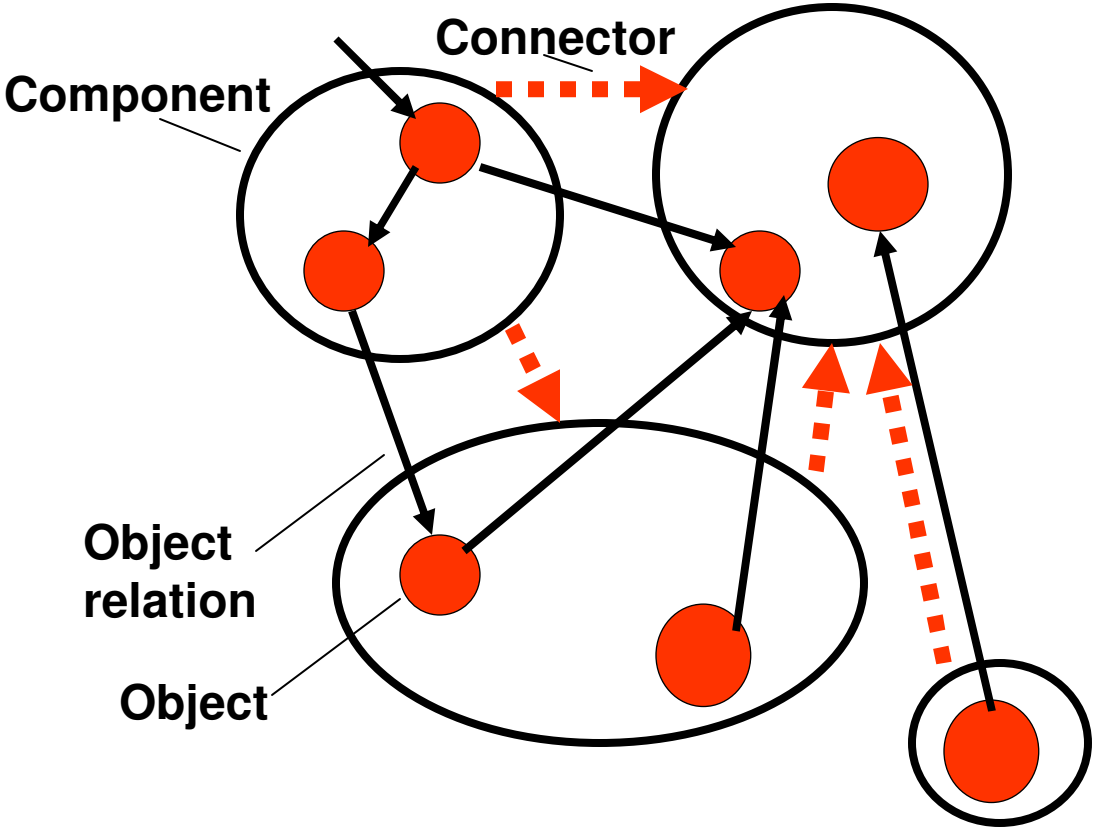



- A node represents a runtime object,
- An edge represents a points-to relation

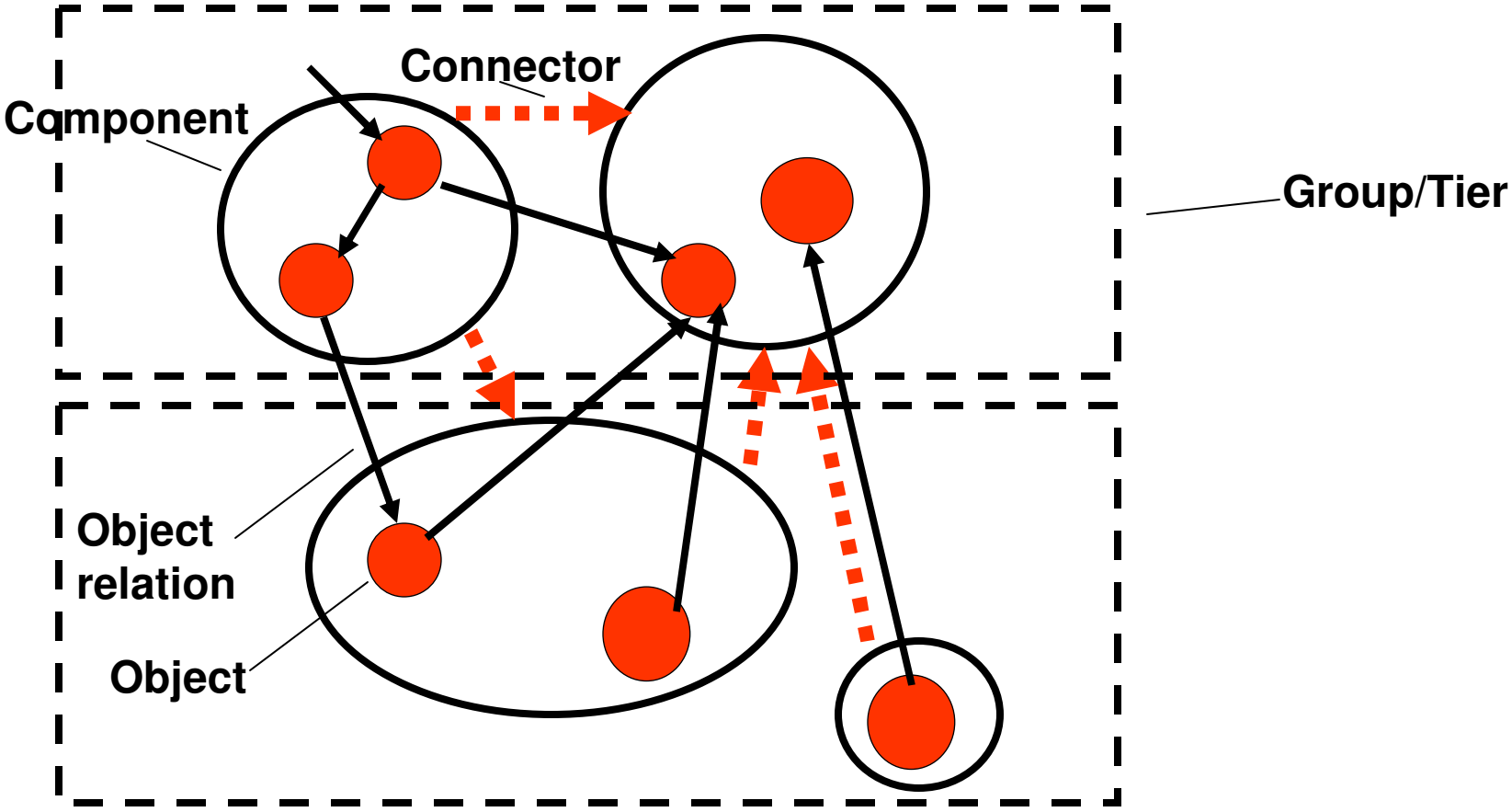
# Abstract objects into “components”



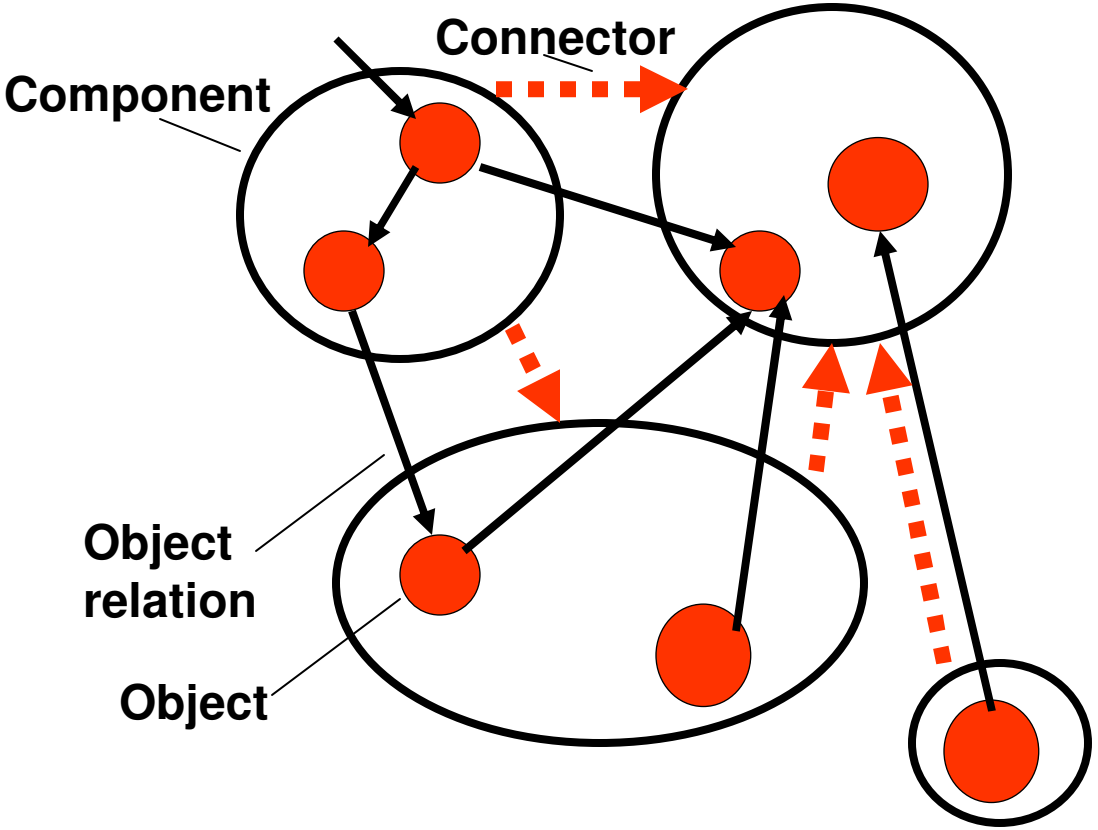
# Abstract relations between components



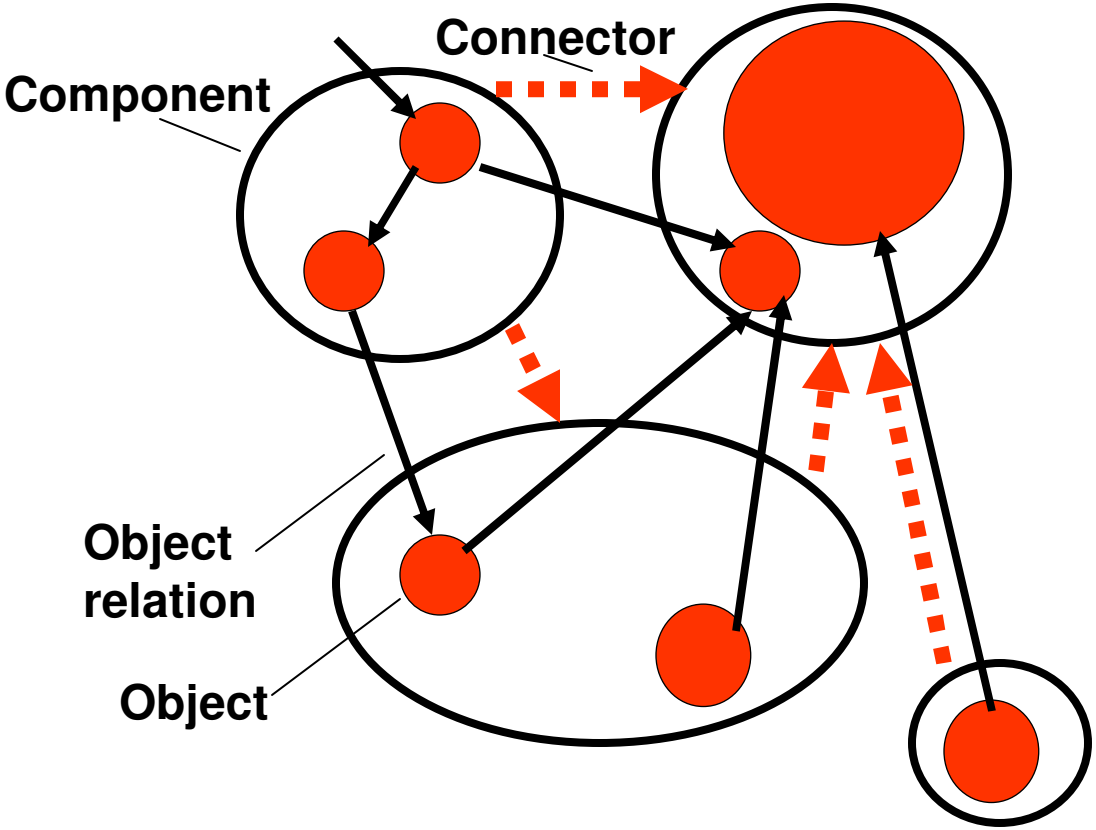
# Organize components into groups/tiers



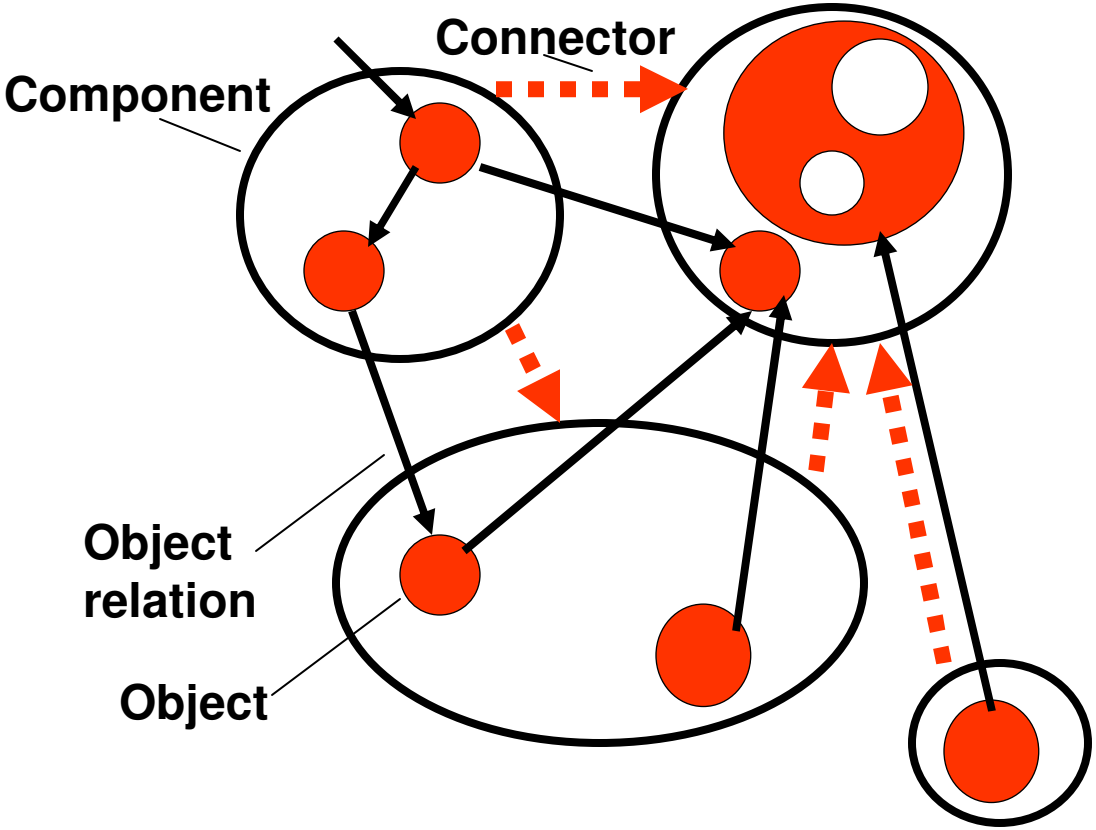
# Make some components part of others



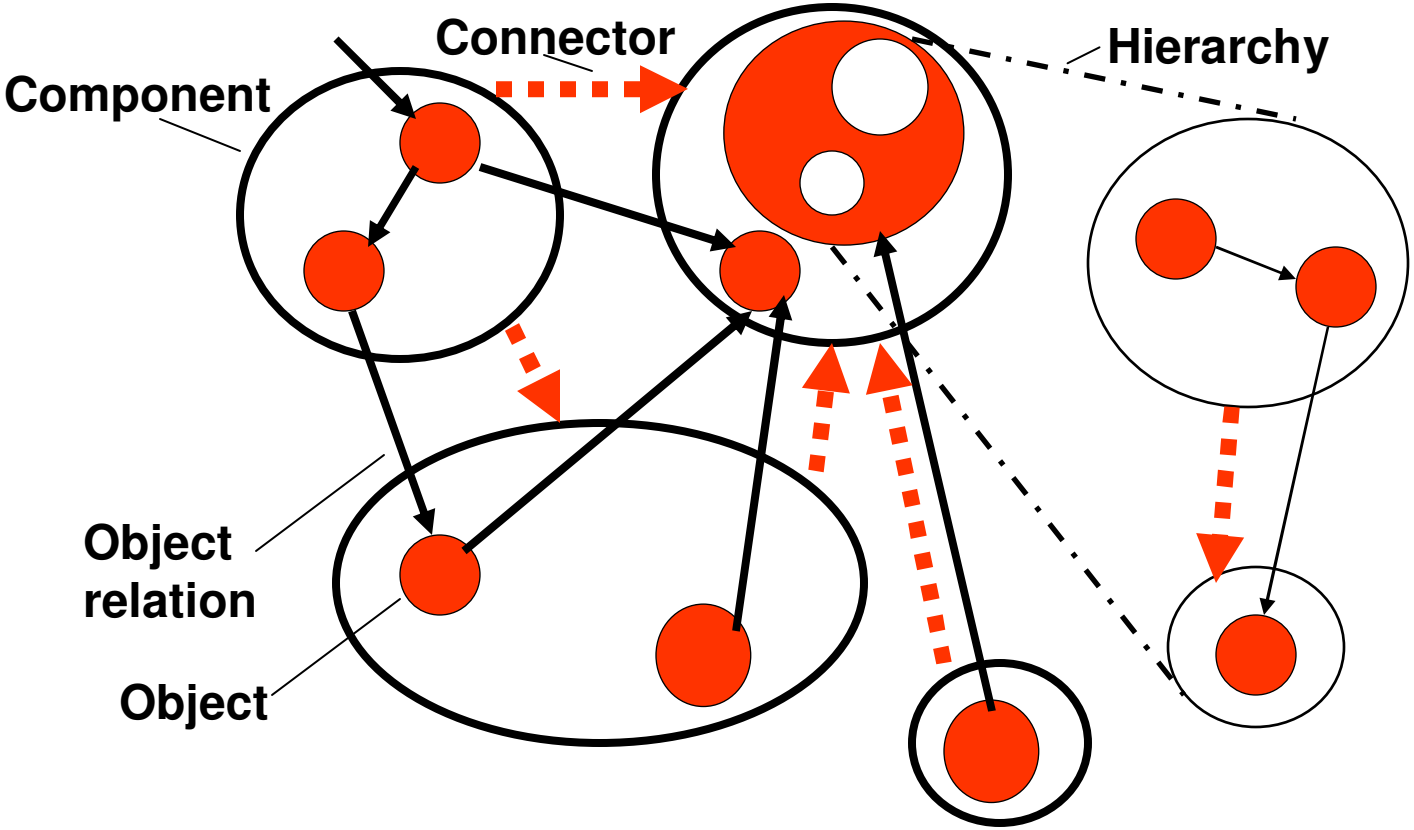
# Make some components part of others



# Make some components part of others



# Make some components part of others

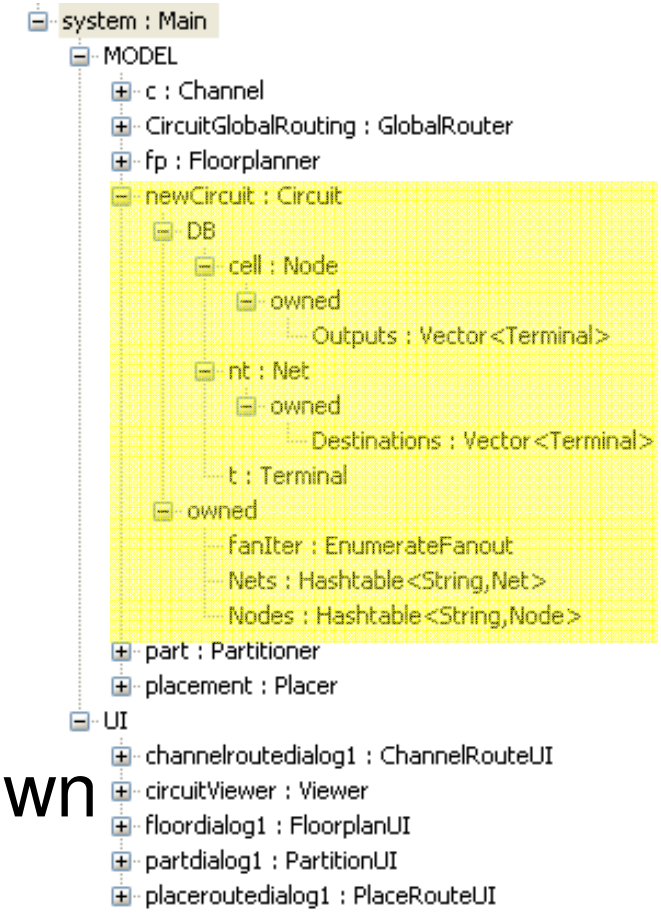


# Central difficulty

Architectural **hierarchy** not readily observable in program written in general purpose programming language

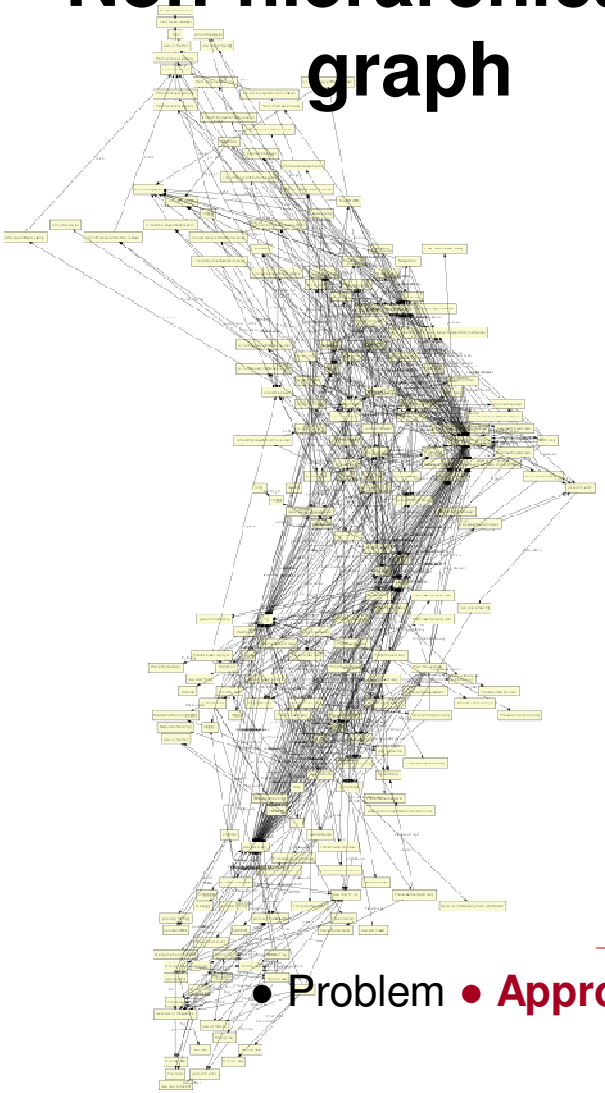
# Key idea: use hierarchy to convey architectural abstraction

- Pick top-level entry point
- Use ownership to impose **conceptual hierarchy**
- Convey **abstraction** by **ownership hierarchy**:
  - Architecturally significant objects near top
  - Low-level objects further down

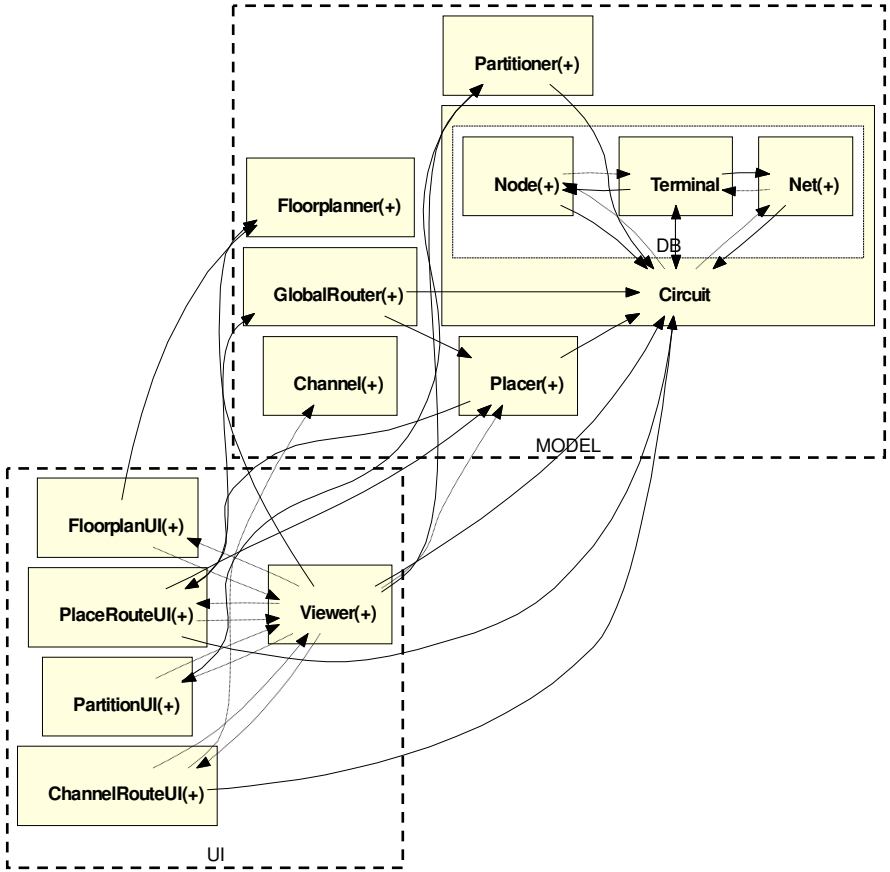


# Collapse objects based on ownership (and types) to achieve abstraction

## Non-hierarchical graph



## Hierarchical graph



# Research contribution: SCHOLIA

**SCHOLIA**: **s**tatic **c**onformance **h**ecking of **o**bject-based structura**l** v**i**ews of **a**rchitecture.

*Scholia are annotations inserted on the margin of an ancient manuscript.*
*The approach supports existing, i.e., legacy systems, and uses annotations.*

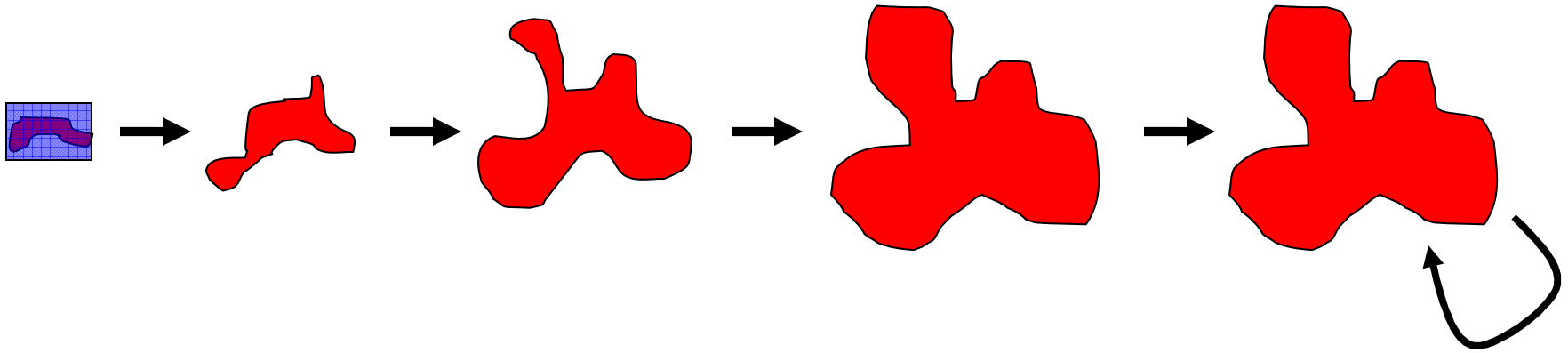
# Key idea: hierarchical object graph extracted statically

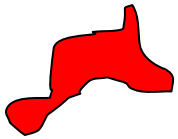
- Extract **global object graph**
  - Convey **architectural abstraction**
  - by **ownership hierarchy;** and
  - (optionally) by **types**
- Use **static analysis**
- Achieve **soundness**

# Key idea: rely on ownership annotations

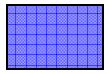
- Rely on **local, modular, statically type-checkable ownership annotations**
  - Use **language support** for annotations
  - Minimally invasive hints about architecture
  - Do not require new language or library
- Follow **extract-abstract-analyze** model

# Review: soundness and precision

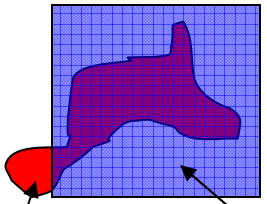




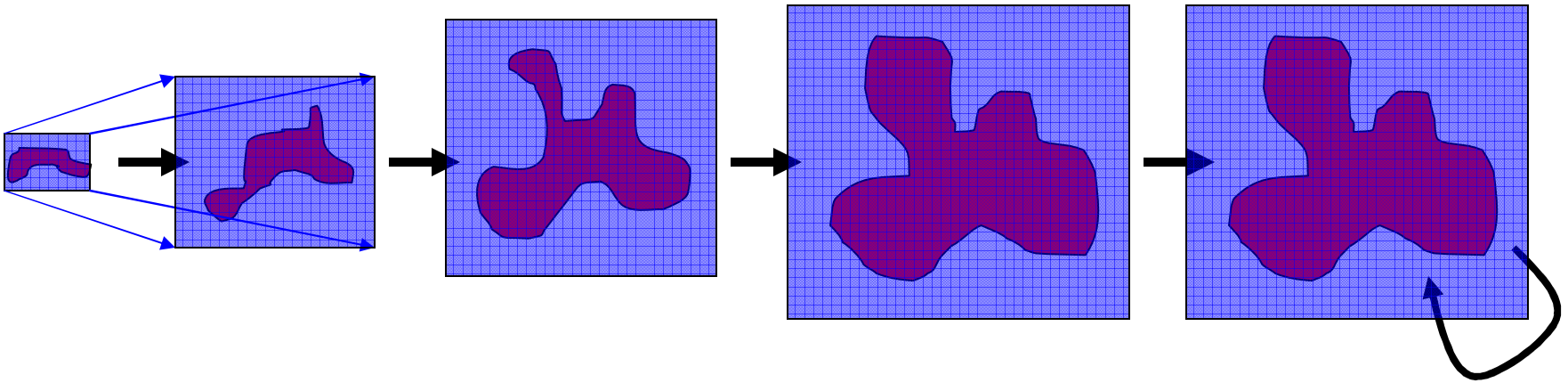
Program state covered in actual execution

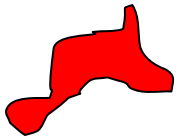


Program state covered by abstract interpretation with analysis

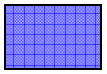


# Review: soundness and precision

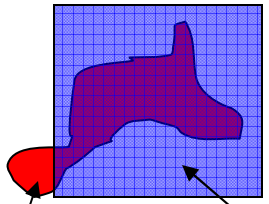




Program state covered in actual execution

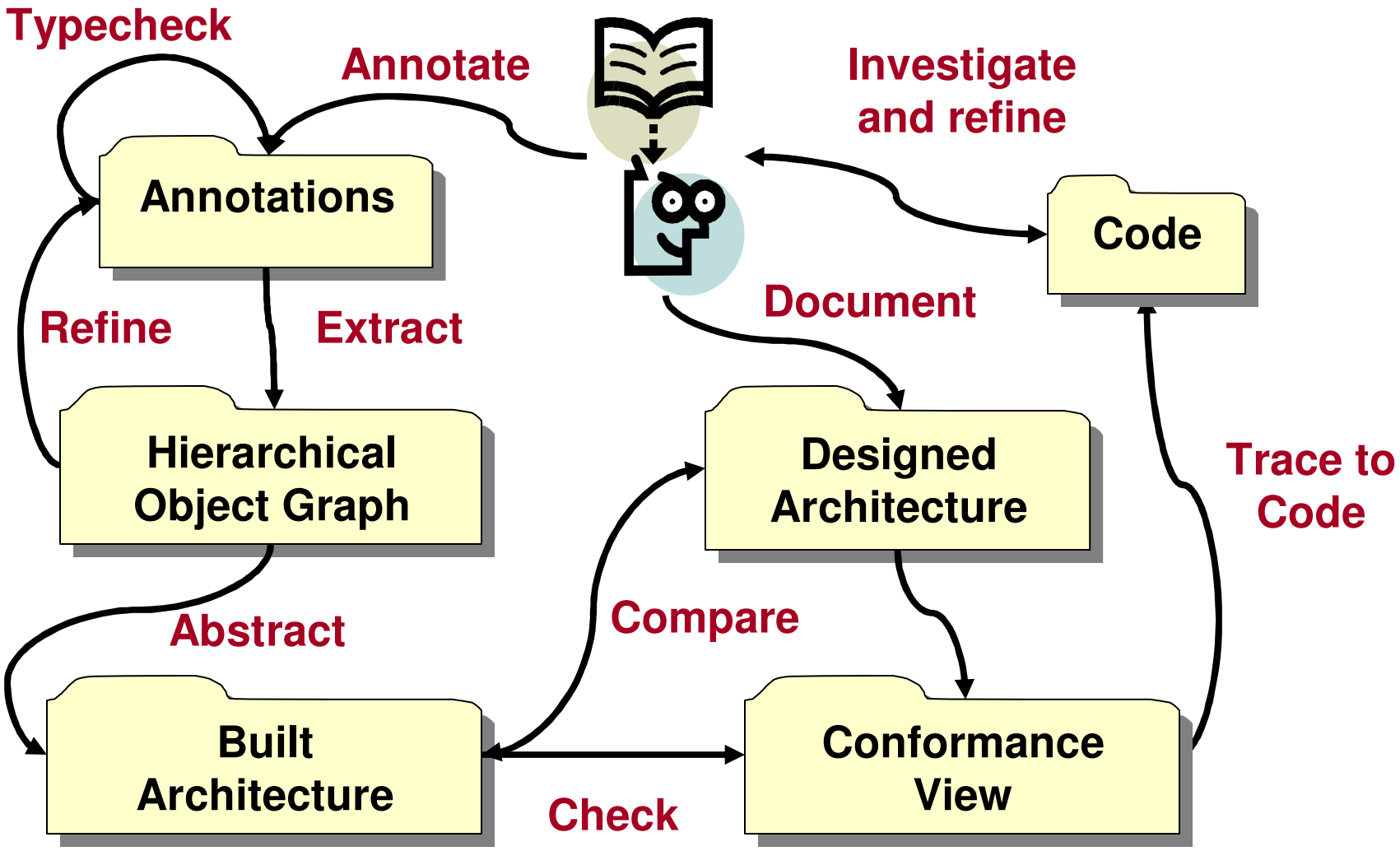


Program state covered by abstract interpretation with analysis

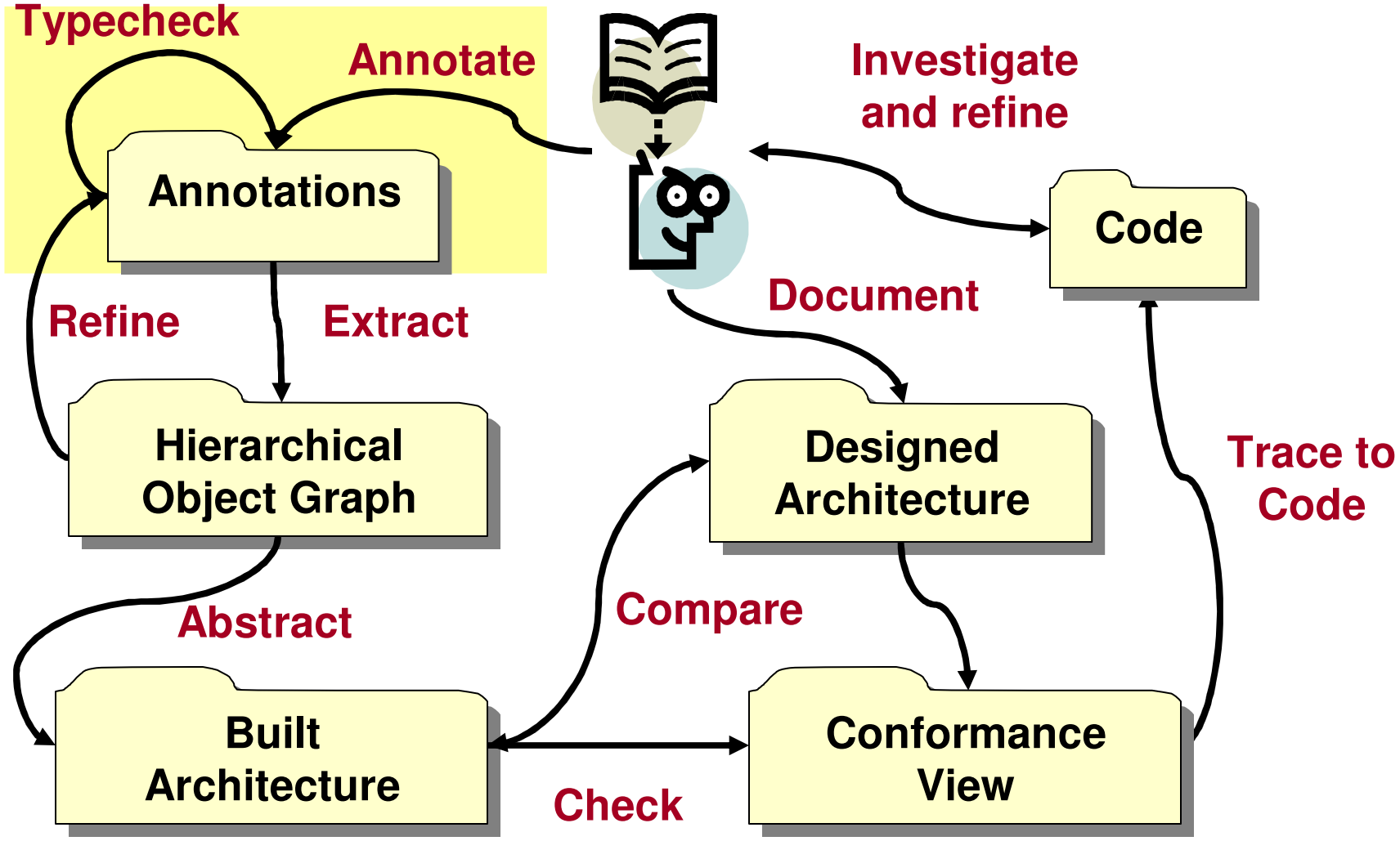


# SCHOLIA'S extract-abstract-check strategy

modeled closely after Reflexion Models [Murphy et al., TSE'01]

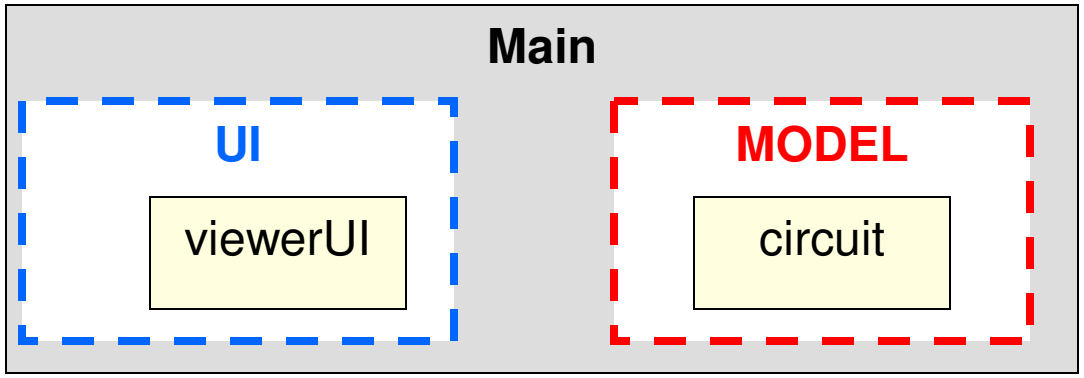


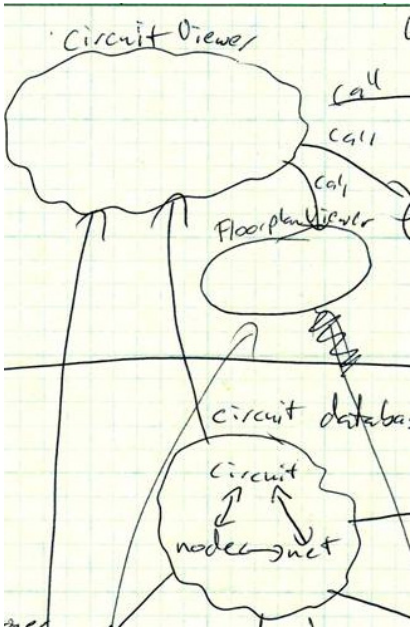
# SCHOLIA: annotate + typecheck



# Ownership domains are groups of objects

[Aldrich and Chambers, ECOOP'04] [Krishnaswami and Aldrich, PLDI'05]




```
class Main {
  domain UI, MODEL;

  UI Viewer viewerUI;
  MODEL Circuit circuit;
  ...
}
```

object: Type — Object

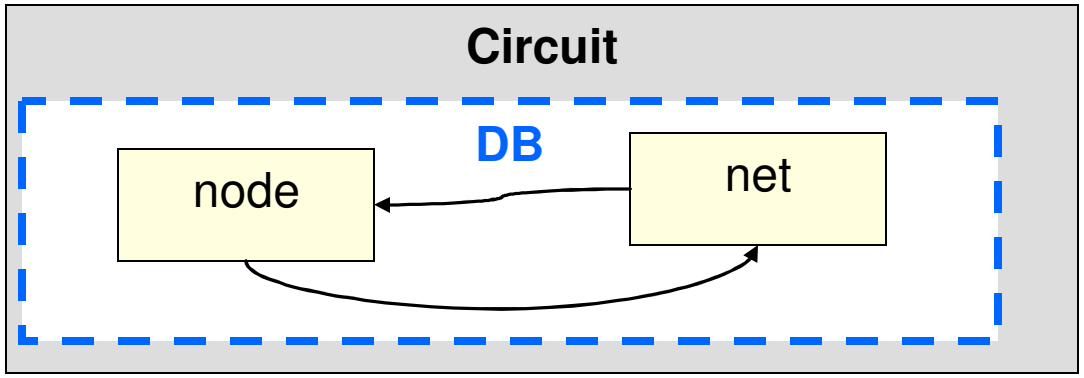
Type — Type

*Declarations are simplified*

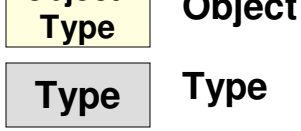
- Ownership domain = conceptual group of objects
- Each object **in exactly one domain**

# Each class can declare domains

[Aldrich and Chambers, ECOOP'04] [Krishnaswami and Aldrich, PLDI'05]

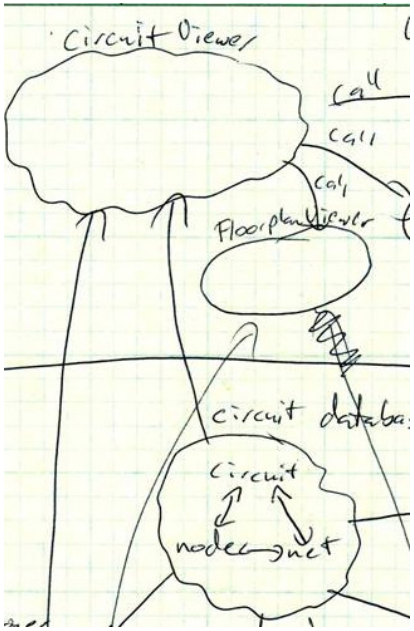



```
class Circuit {
  public domain DB;

  DB Node node;
  DB Net net;
  ...
}
```




Object

Type



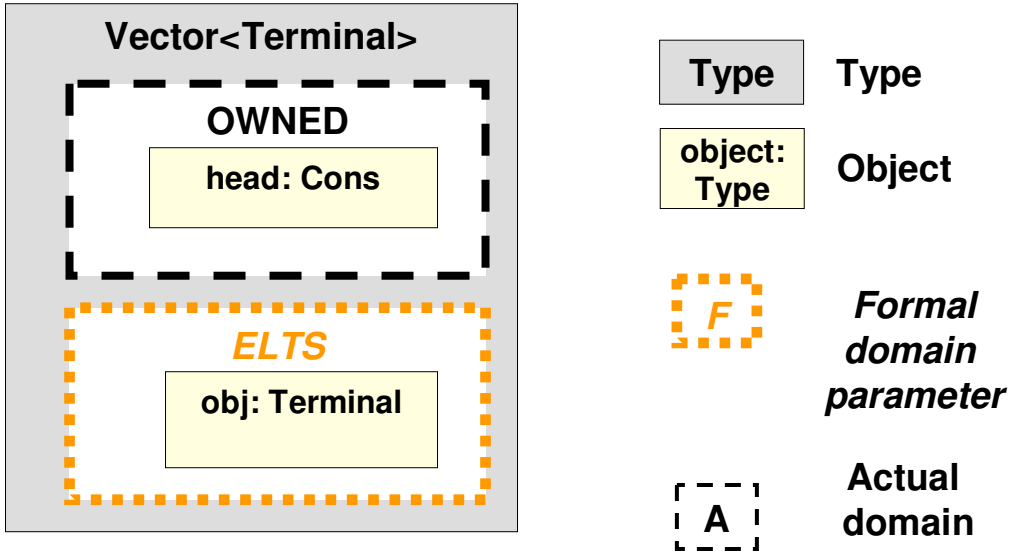

*Declarations are simplified*

# Domain parameters allow state sharing

[Aldrich and Chambers, ECOOP'04] [Krishnaswami and Aldrich, PLDI'05]

- Reusable or library code often parametric with respect to ownership
  - Typically, Vector does not "own" its elements
  - Takes **domain parameter *ELTS*** for elements

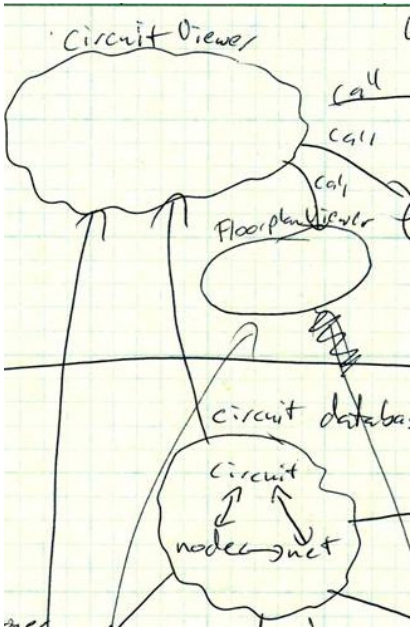
```
class Vector<ELTS> {
  domain OWNED;
  ELTS Terminal obj;
  OWNED Cons head;
  ...
}
class Circuit {
  public domain DB;
  domain OWNED;
  DB Node nd;
  OWNED Vector<DB> nodes;
  ...
}
```



# SCHOLIA's tools use Java 1.5 annotations

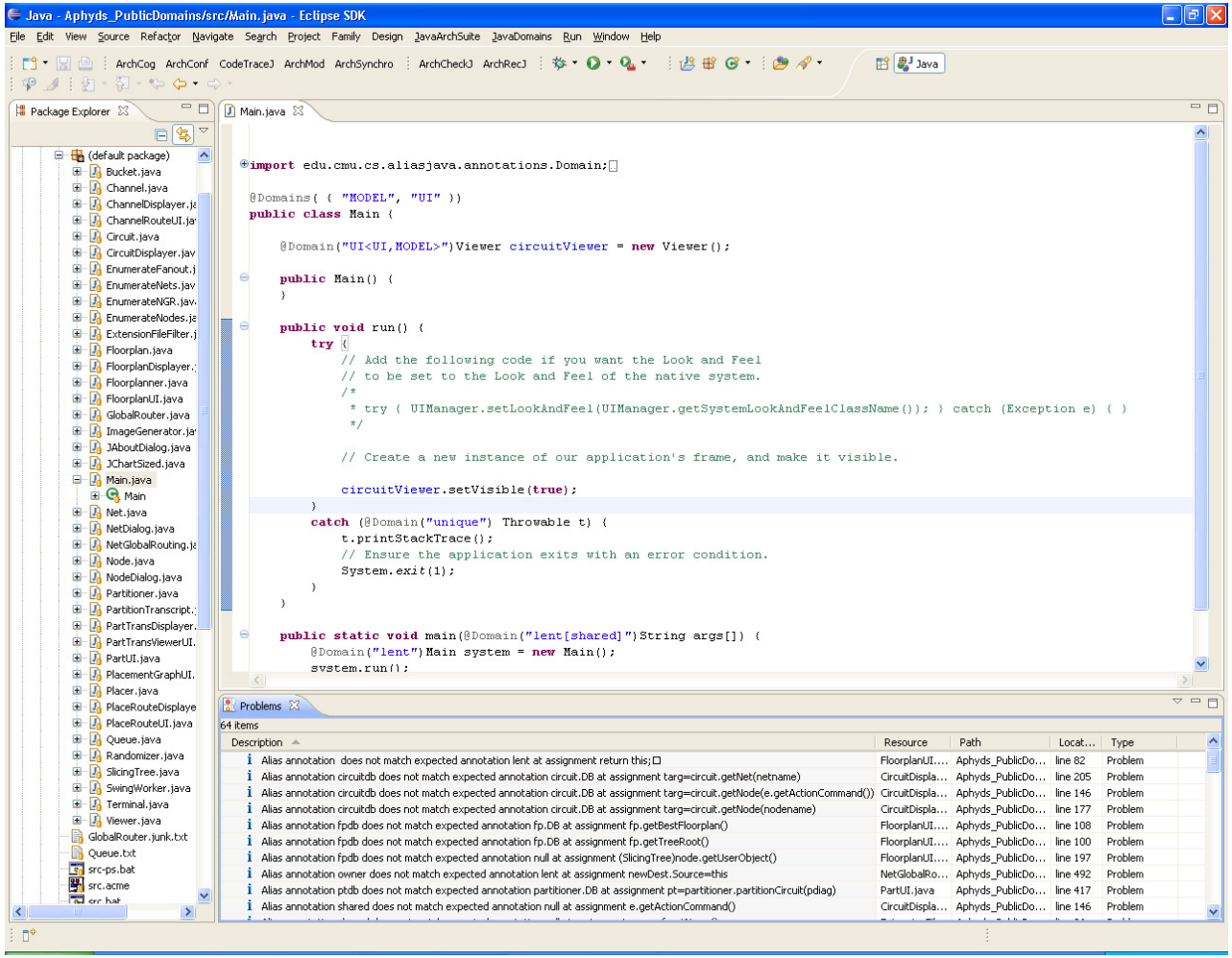
[Abi-Antoun and Aldrich, IWACO'07]

```
@Domains({"UI", "MODEL"})
class Main {
  @Domain("UI") Viewer viewerUI;
  @Domain("MODEL") Circuit circuit;
  ...
}
```

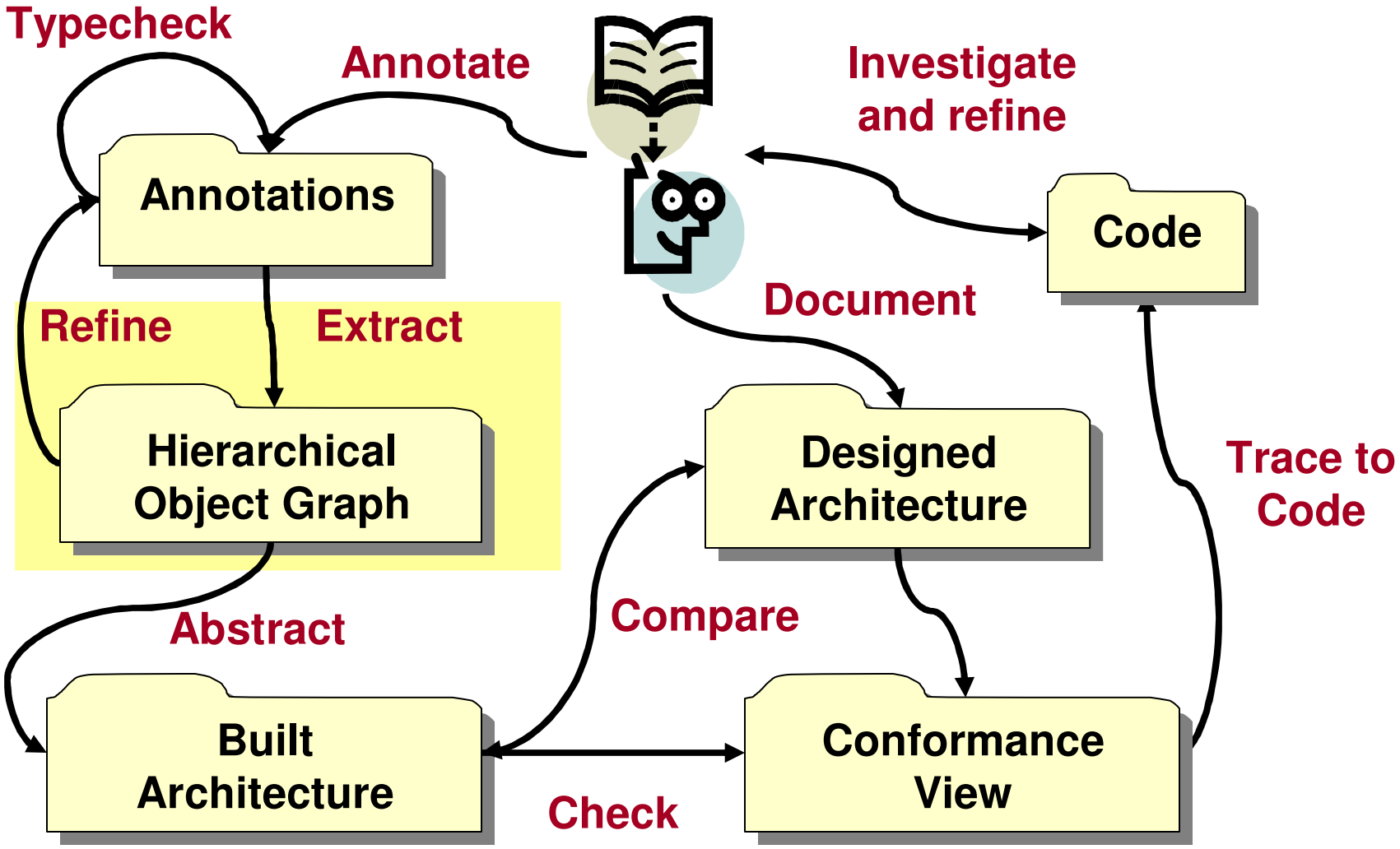



- Tools use **existing language support for annotations** (available in Java 1.5, C#, …)
- Annotations do not change runtime semantics

# ArchCheckJ: check annotations modularly; address warnings



# SCHOLIA: extract object graph



# Generate ObjectGraph by abstract interpretation of program

- **ObjectGraph**: graph of **objects** and **domains** (no types/classes)
  - Analyze **local, modular** annotations
  - Generate **global object graph**
  - Start from a root class
- Abstractly interpret/execute program:
  - New expression → **Object**
  - Domain declaration → **Domain**
  - Field declaration → **Edge**
- A kind of a points-to analysis

# Challenge: ObjectGraph must show all objects in each domain

- At runtime, each domain parameter bound to some actual domain
  - Track to which the receiver **this** maps
  - Track how a **formal domain parameter maps to (→) actual domain**

```
[this → c.DB.nd]
Bindings := [Node::OWNER → c.DB ]
class Node<OWNER> {
}
```

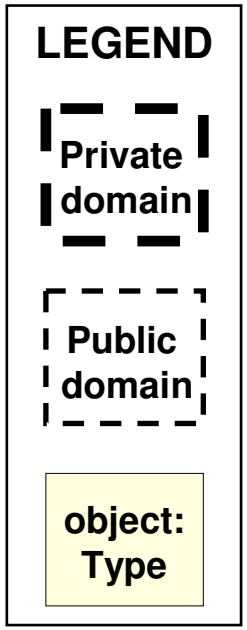
# Challenge: must handle possible aliasing

- Ideally, use an alias analysis
- Here, rely on precision about aliasing from ownership domain annotations:
  - Objects in **different domains cannot alias**
  - Objects in **same domain *may* alias**
  - Offers **adequate precision** when objects have precise declared types

# ObjectGraph: data type declarations

- **OGraph**
  - D ::= **ODomain**(Id = $D_{id}$, Domain = C::d)
  - O ::= **OObject**(Id = $O_{id}$, Owner = D, Type = C)
  - E ::= **OEdge**(From = $O_{src}$, To = $O_{dst}$)
- Here, declarations are simplified
  - OObject also has domain parameters $D_i$
  - See paper/dissertation for full details

# ObjectGraph: abstractly interpret new expression into OObject c

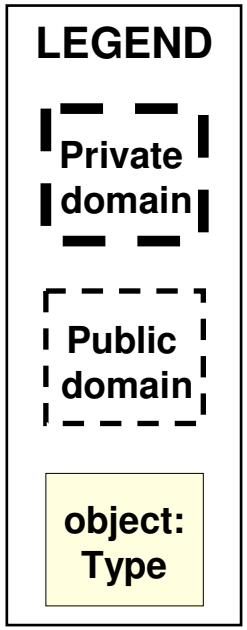




```
Circuit c = new Circuit();
OObject(c, null, Circuit) (O0)
```

```
class Circuit {
...
}
```

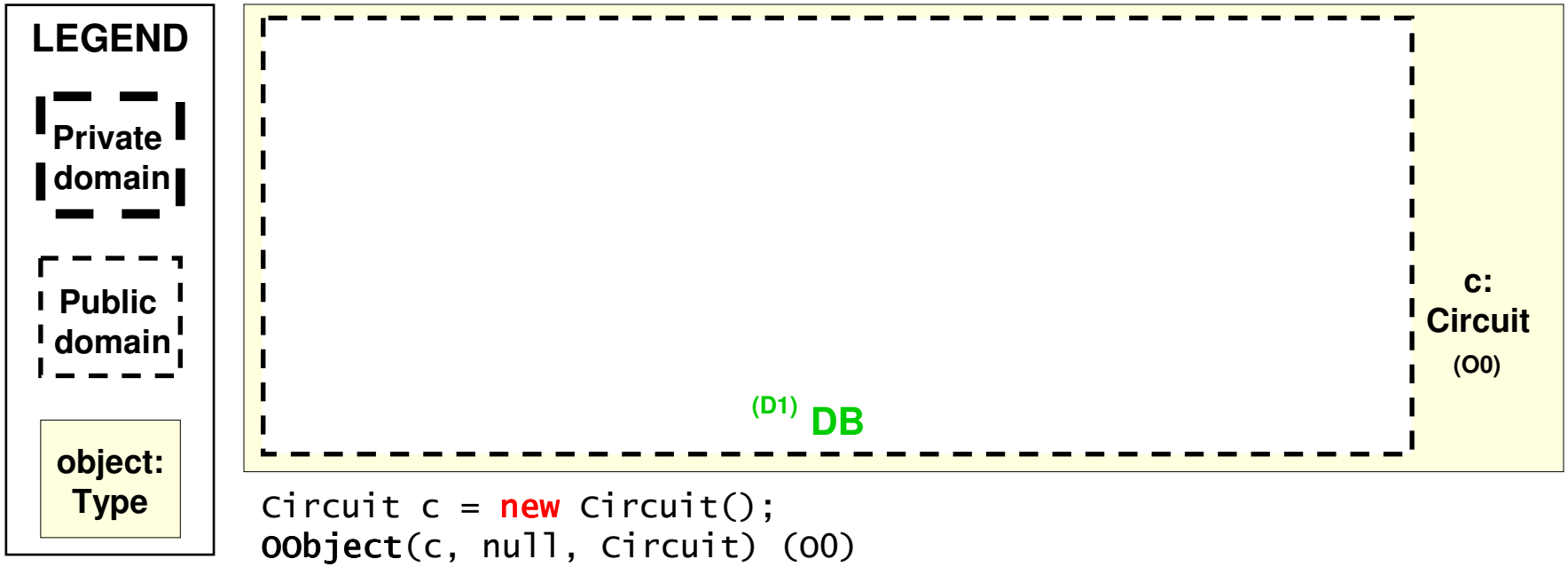
# ObjectGraph: analyze class Circuit in the context of OObject c





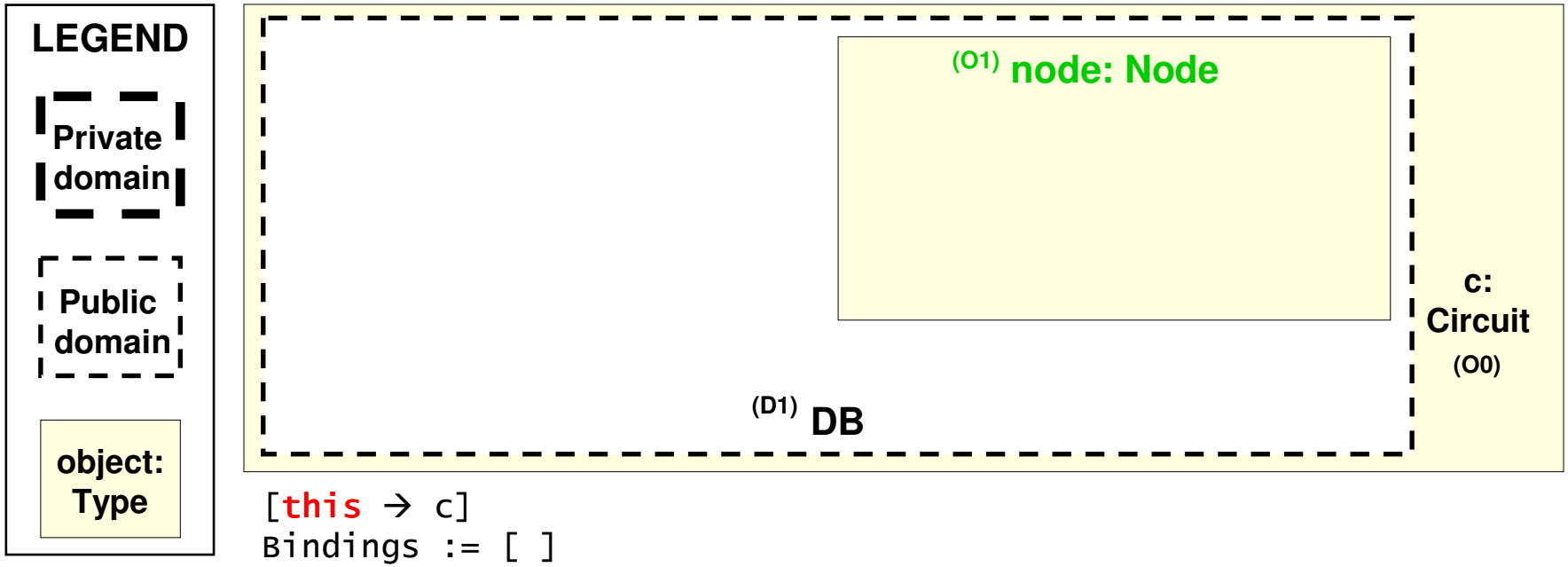
```
Circuit c = new Circuit();
OObject(c, null, Circuit) (O0)
analyze(Circuit, c, [ ])

class Circuit {
...
}
```

# ObjectGraph: abstractly interpret domain declaration into ODomain c.DB



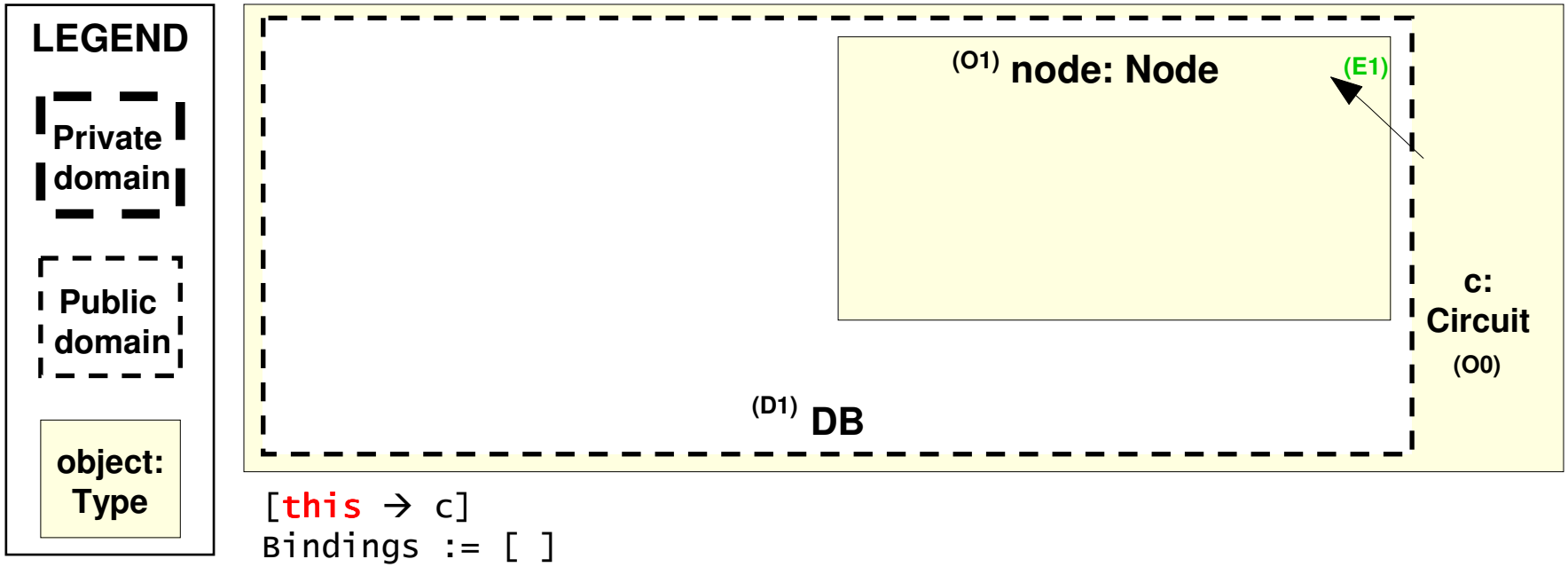

```
Circuit c = new Circuit();
OObject(c, null, Circuit) (O0)

[this → c]
Bindings := [ ]
class Circuit {
  public domain DB;
  ODomain(c.DB, Circuit::DB) (D1)
  ...
}
```

# ObjectGraph: abstractly interpret new expression into OObject c.DB.nd



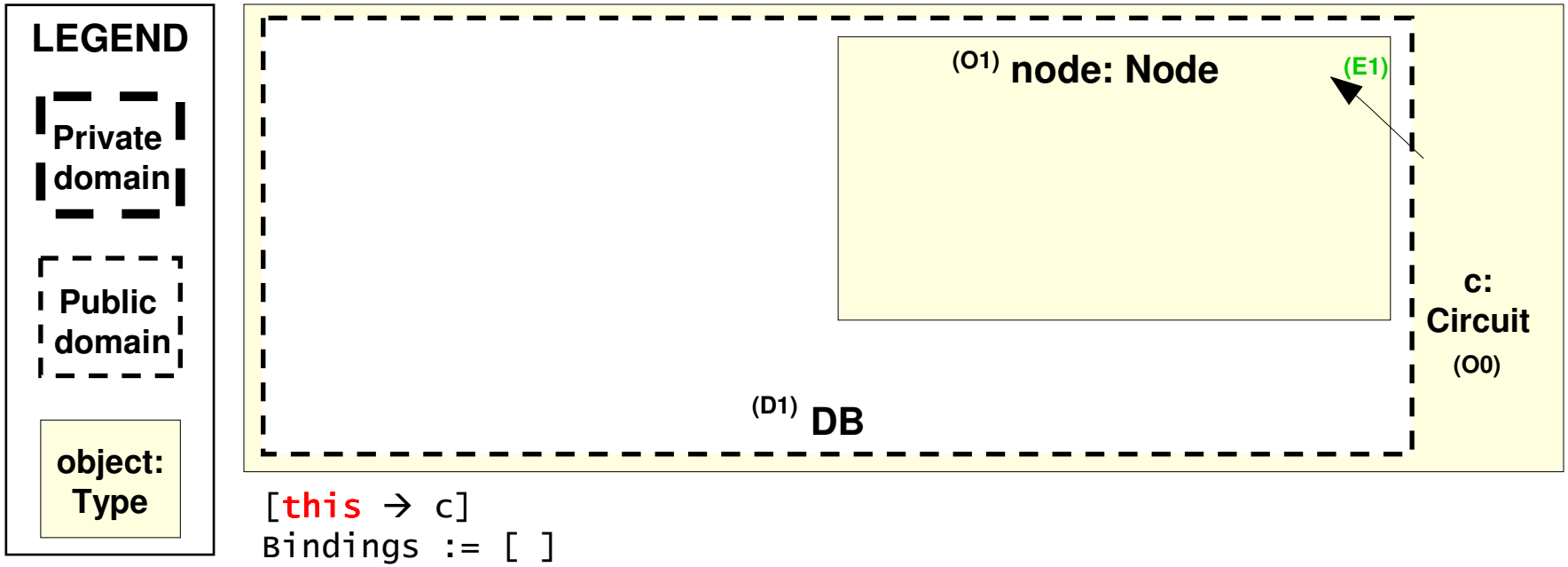

```
[this → c]
Bindings := [ ]
class Circuit {
  ...
  DB Node nd = new Node<DB>();
  OObject(c.DB.nd, c.DB, Node) (O1)
  ...
}
```

# ObjectGraph: abstractly interpret field declaration into OEdge



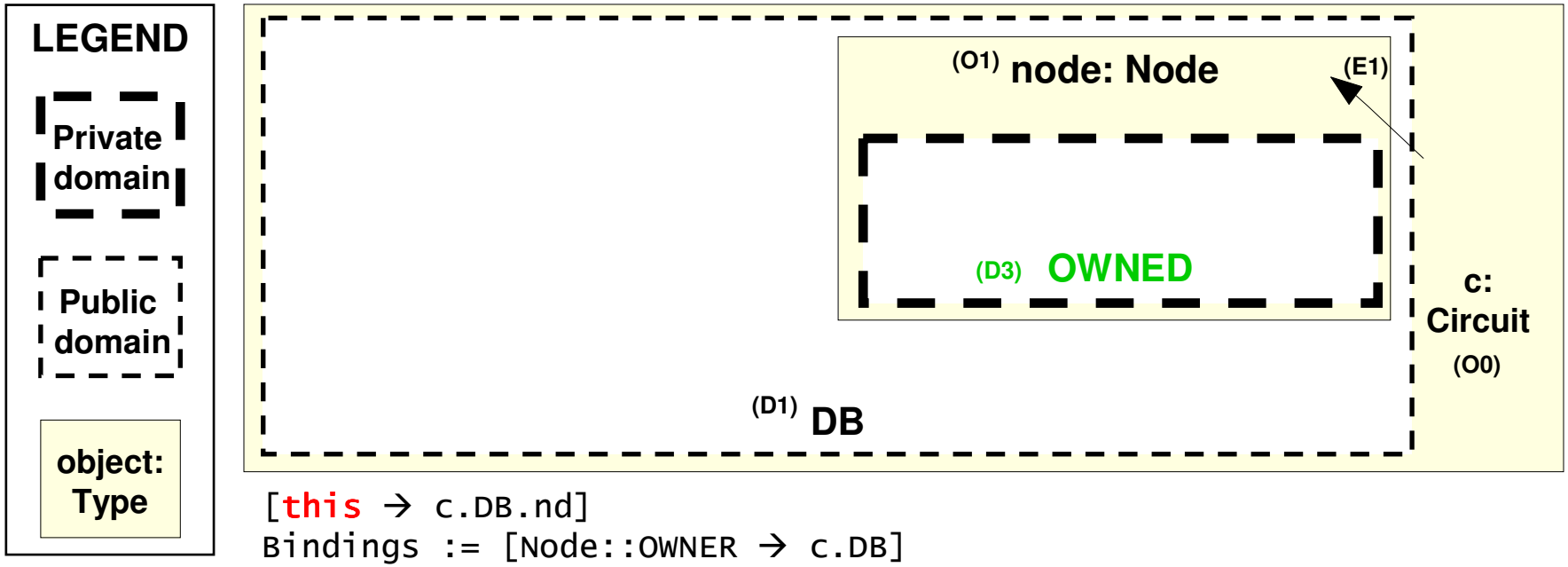

```
[this → c]
Bindings := [ ]
class Circuit {
  ...
  DB Node nd = new Node<DB>();
  OObject(c.DB.nd, c.DB, Node) (O1)
  OEdge(c, c.DB.nd) (E1)
  ...
}
```

# ObjectGraph: analyze class Node in context of OObject c.DB.nd



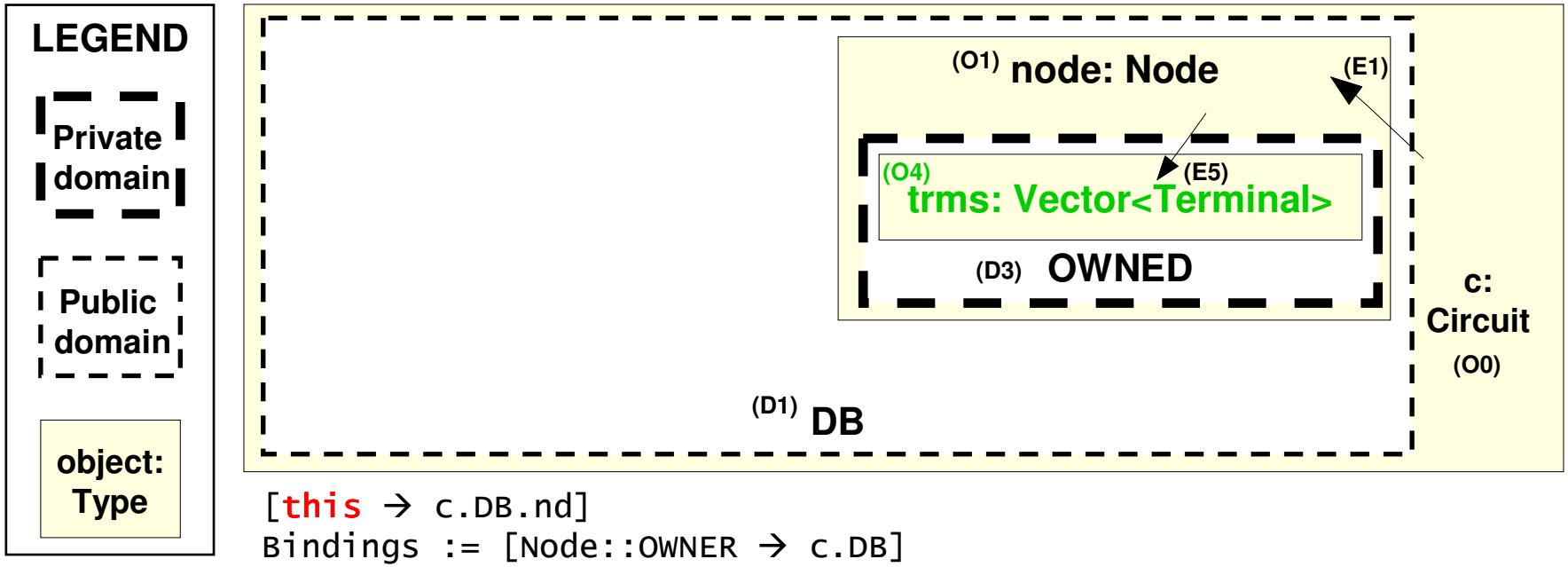

```
[this → c]
Bindings := [ ]
class Circuit {
  ...
  DB Node nd = new Node<DB>();
  analyze(Node, c.DB.nd, [Node::OWNER→ c.DB])
  ...
}
class Node<OWNER> {
}
```

# ObjectGraph: abstractly interpret domain declaration into ODomain



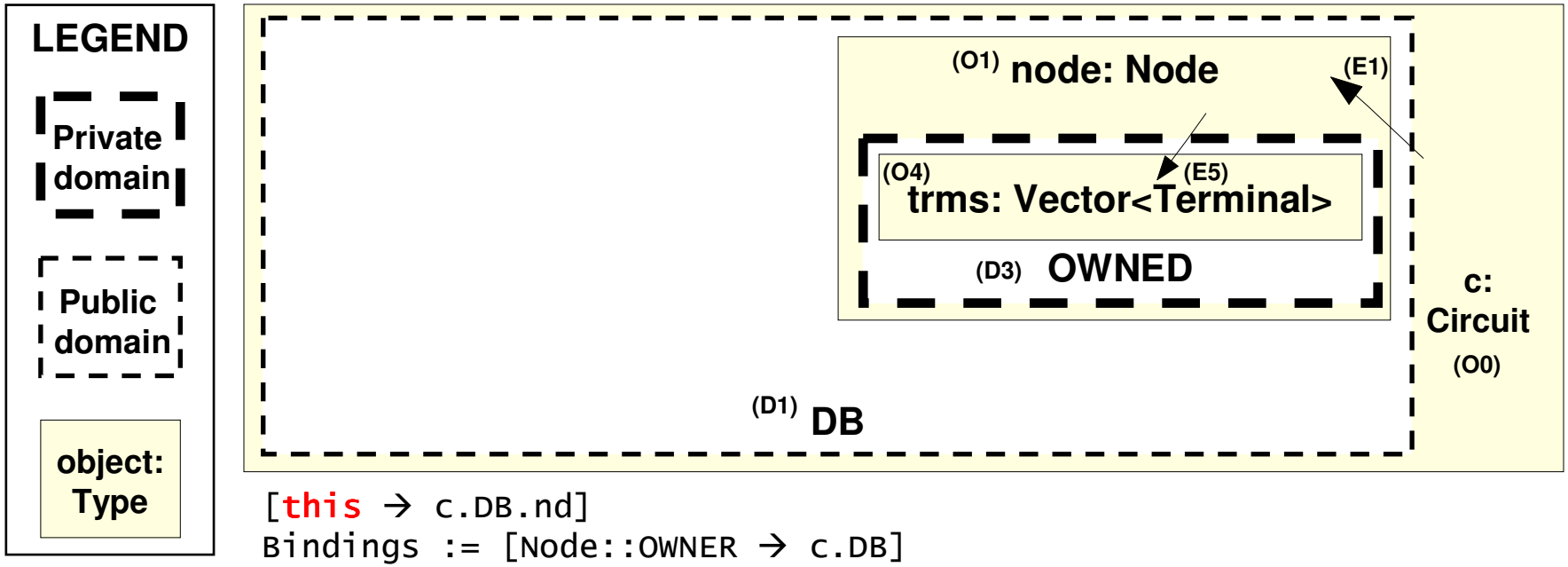

```
[this → c.DB.nd]
Bindings := [Node::OWNER → c.DB]
class Node<OWNER> {
 ODomain(c.DB.nd.OWNED, Node::OWNED) (D3)
 domain OWNED;
 ...
}
```

# ObjectGraph: abstractly new expression into OObject c.DB.nd.OWNED.trms



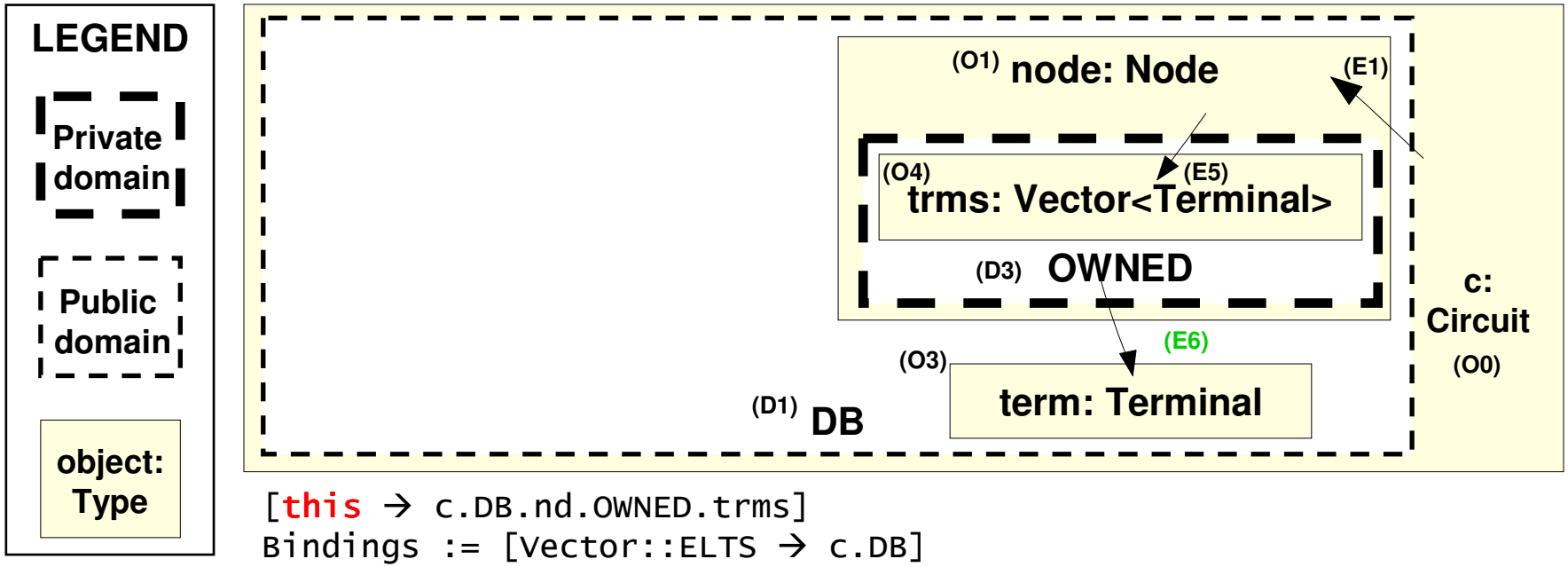

```
[this → c.DB.nd]
Bindings := [Node::OWNER → c.DB]
class Node<OWNER> {
 ...
 OObject(c.DB.nd.OWNED.trms, c.DB.nd.OWNED, Vector<Terminal>) (O4)
 OWNED Vector<OWNER Terminal> trms = new Vector<...>();
 OEdge(c.DB.nd, c.DB.nd.OWNED.trms) (E5)
 ...
}
```

# ObjectGraph: analyze class Vector in the context of OObject c.DB.nd.OWNED.trms




```
[this → c.DB.nd]
Bindings := [Node::OWNER → c.DB]
class Node<OWNER> {
  ...
  OWNED Vector<OWNER Terminal> trms = new Vector<...>();
  analyze(Vector, c.DB.nd.OWNED.trms, [Vector::ELTS → c.DB])
 ...
}
class Vector<ELTS T> {
}
```

# ObjectGraph: abstractly interpret field declaration into OEdge

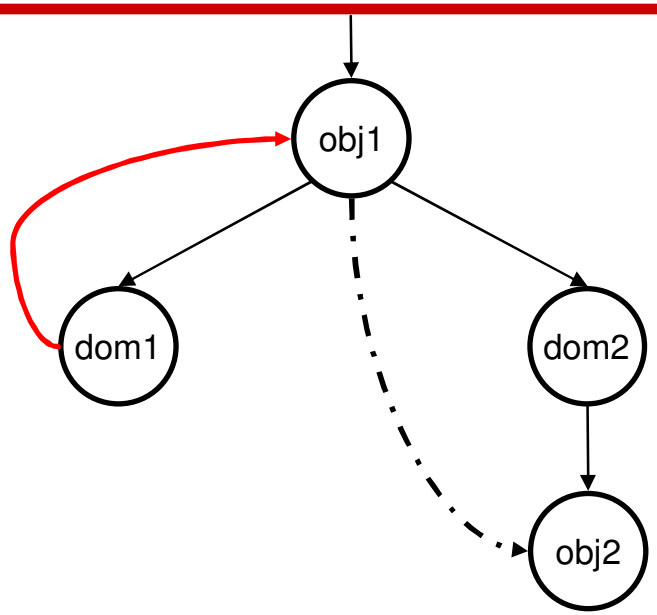



```
[this → c.DB.nd.OWNED.trms]
Bindings := [Vector::ELTS → c.DB]
[T → Terminal
class Vector<ELTS T> {
 OObject(c.DB.term, Terminal) in lookup(c.DB, Terminal)
 OEdge(c.DB.nd.OWNED.trms, c.DB.term) (E6)
 ELTS T obj;
}
```

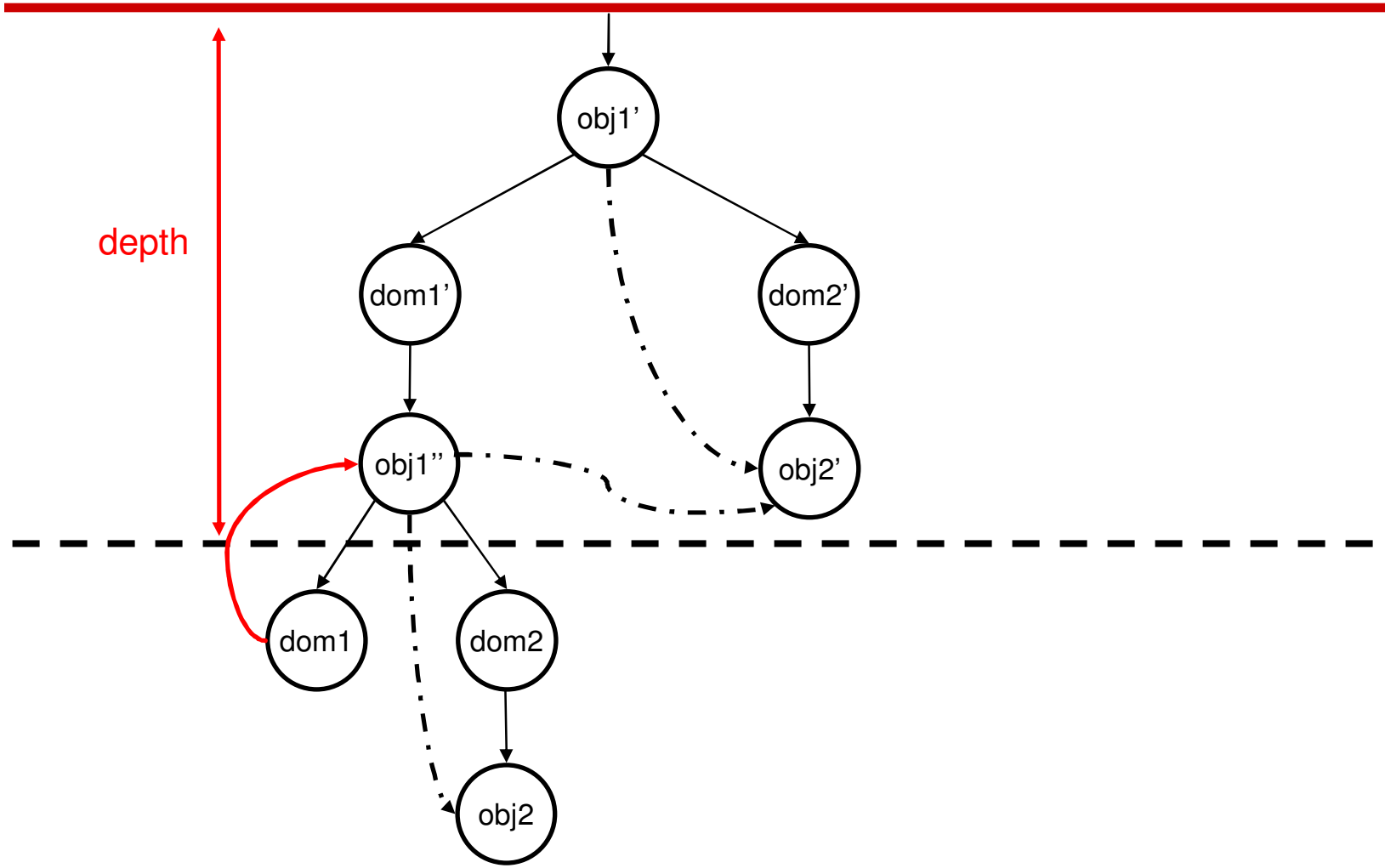
# Challenge: ObjectGraph can have cycles. Unfold it for visualization (DisplayGraph)

- Recursive types create cycles in the ObjectGraph
  - This avoids non-termination
  - ODomain does not have a unique owning OObject
  - Details in paper/dissertation
- Visualization unfolds **ObjectGraph** to limited depth

# Unfold ObjectGraph to limited depth, for visualization, as a DisplayGraph



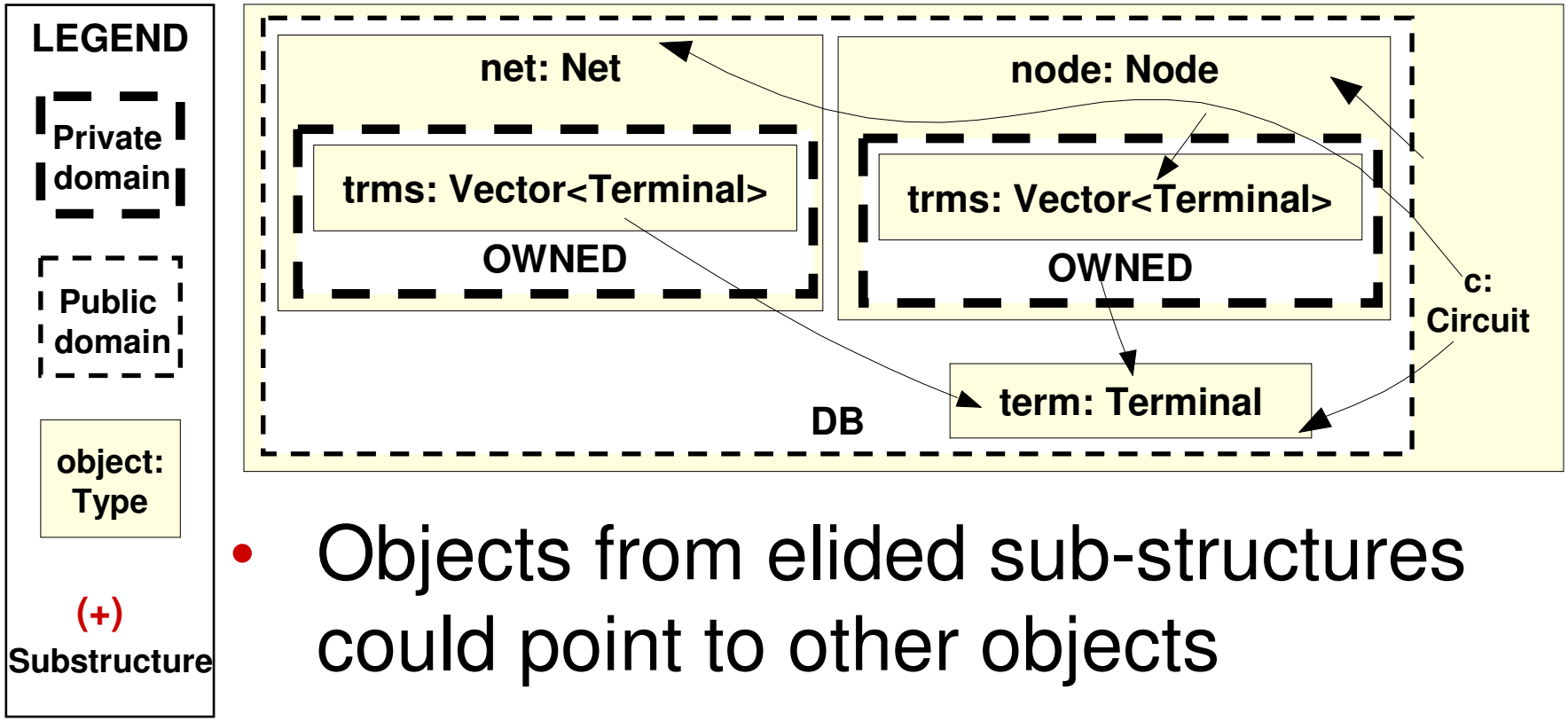
# Unfold ObjectGraph to limited depth, for visualization, as a DisplayGraph



# Developer interacts with DisplayGraph

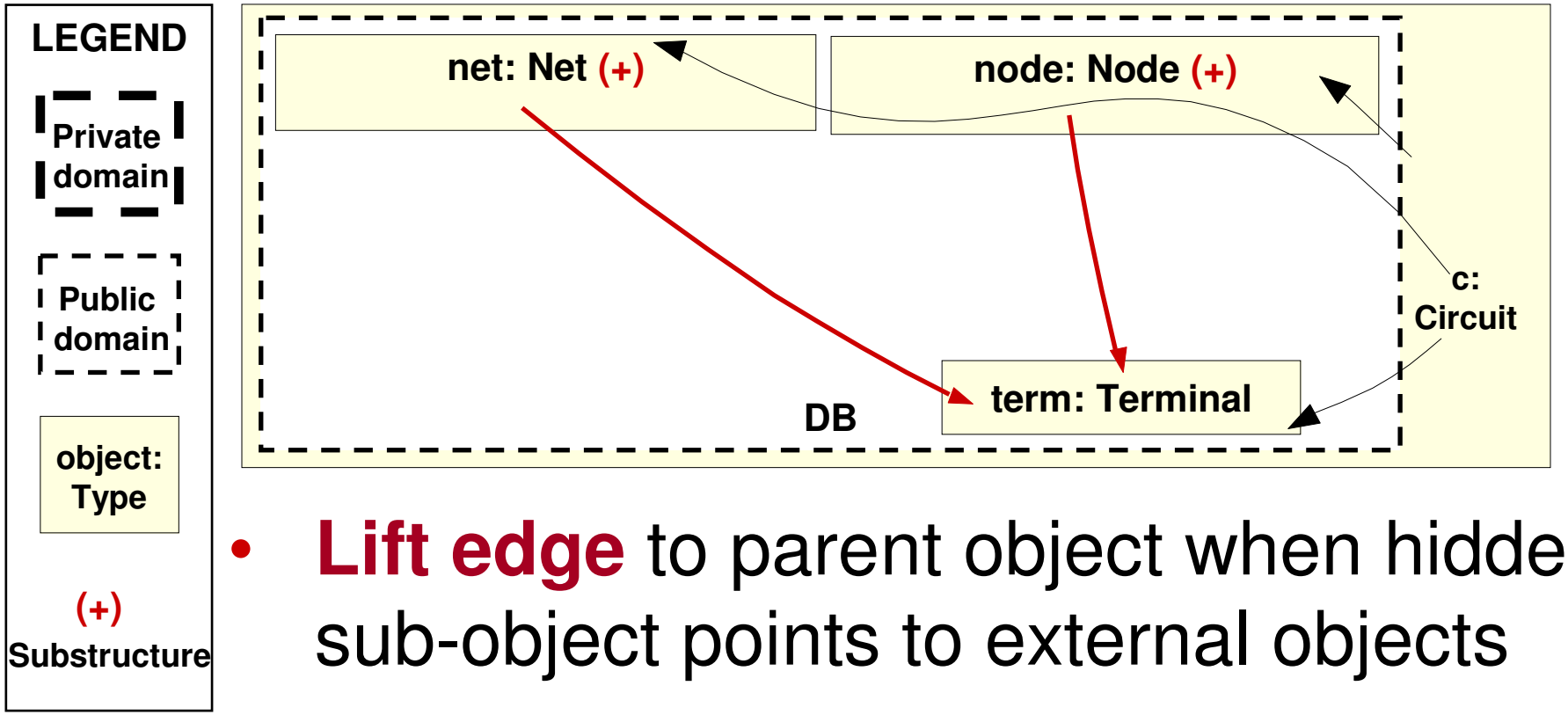
- Control unfolding depth
- Collapse/expand selected elements
- Control abstraction by types

# Expand/collapse objects




- Objects from elided sub-structures could point to other objects

# Expand/collapse objects




- **Lift edge** to parent object when hidden sub-object points to external objects

# Extraction key property: soundness

- **Map each object** to **exactly one** node
- Show **all edges** between objects

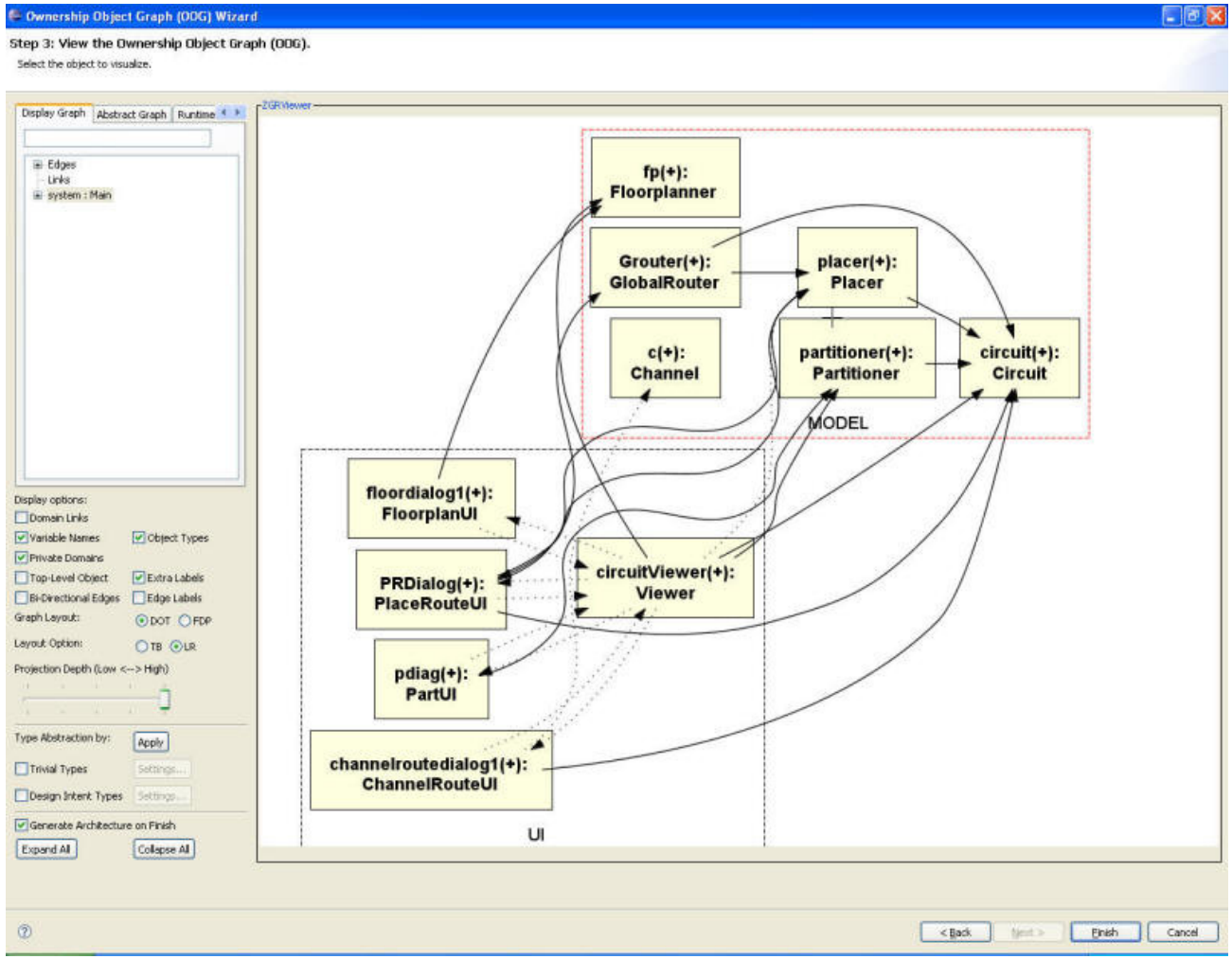
# Demonstrating soundness

- Featherweight Java [Igarashi, Pierce and Wadler, TOPLAS'01 ] + **ownership domains** [Aldrich and Chambers, ECOOP'04]
- **Constraint-based specification**
- **Soundness proof**
  - **Instrumented runtime semantics**
  - **Approximation relation** between runtime states and ObjectGraph
  - Standard **Progress**, **Preservation** theorems
  - Details in dissertation

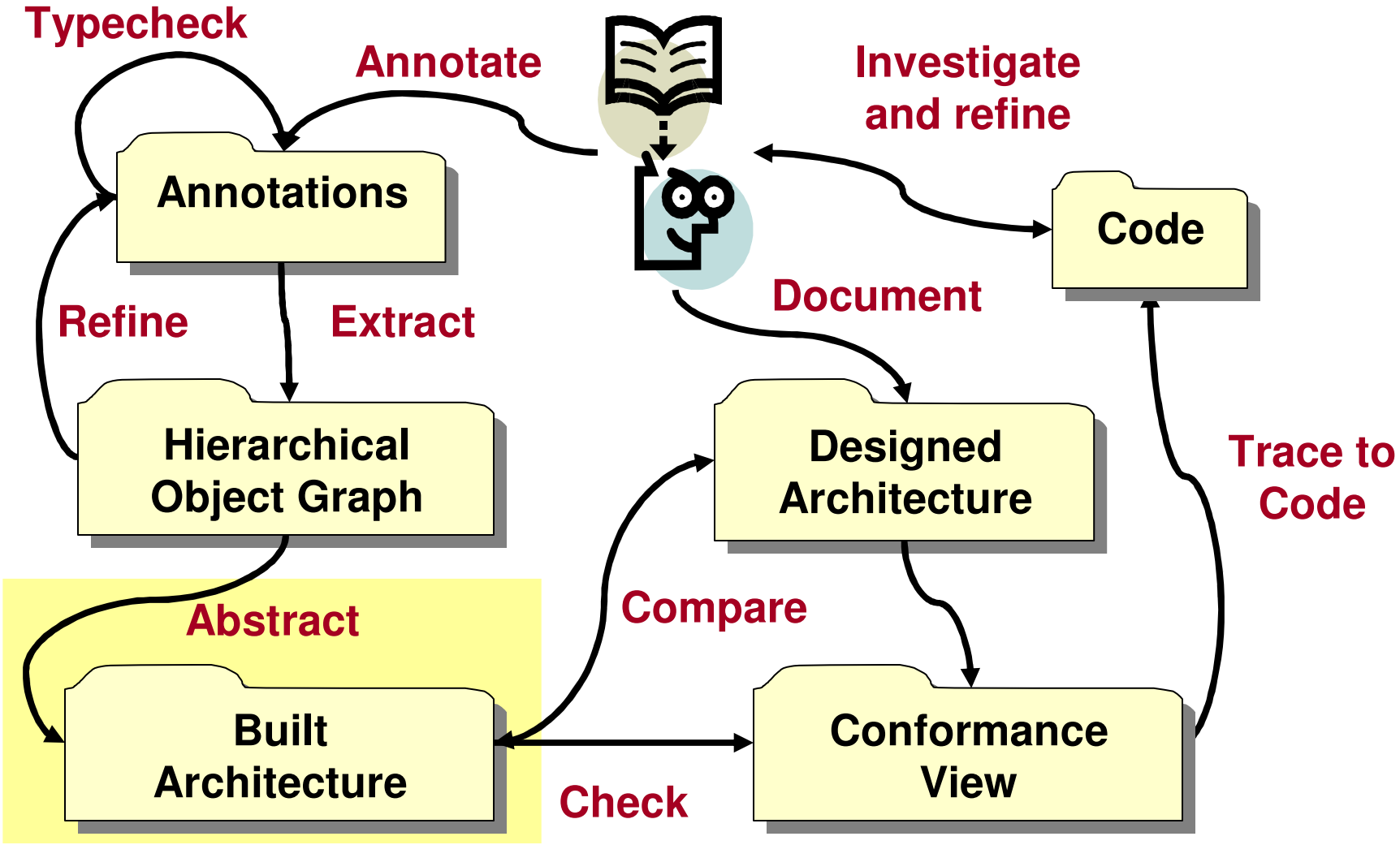
# Inductive argument for soundness

- Start program with abstract state for all possible initial concrete states
- At each step, ensure new abstract state covers all concrete states that could result from executing statement on any concrete state from previous abstract state
- Once no new abstract states are reachable, by induction all concrete program executions have been considered

# ArchRecJ: extract object graph



# SCHOLIA: abstract object graph

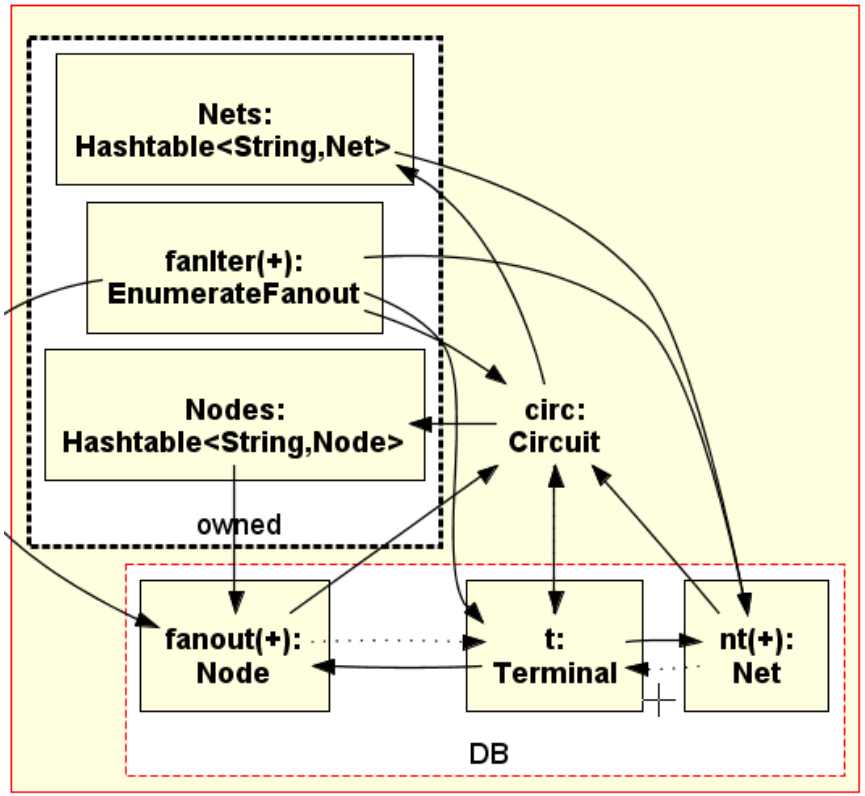


# Why abstract an object graph?

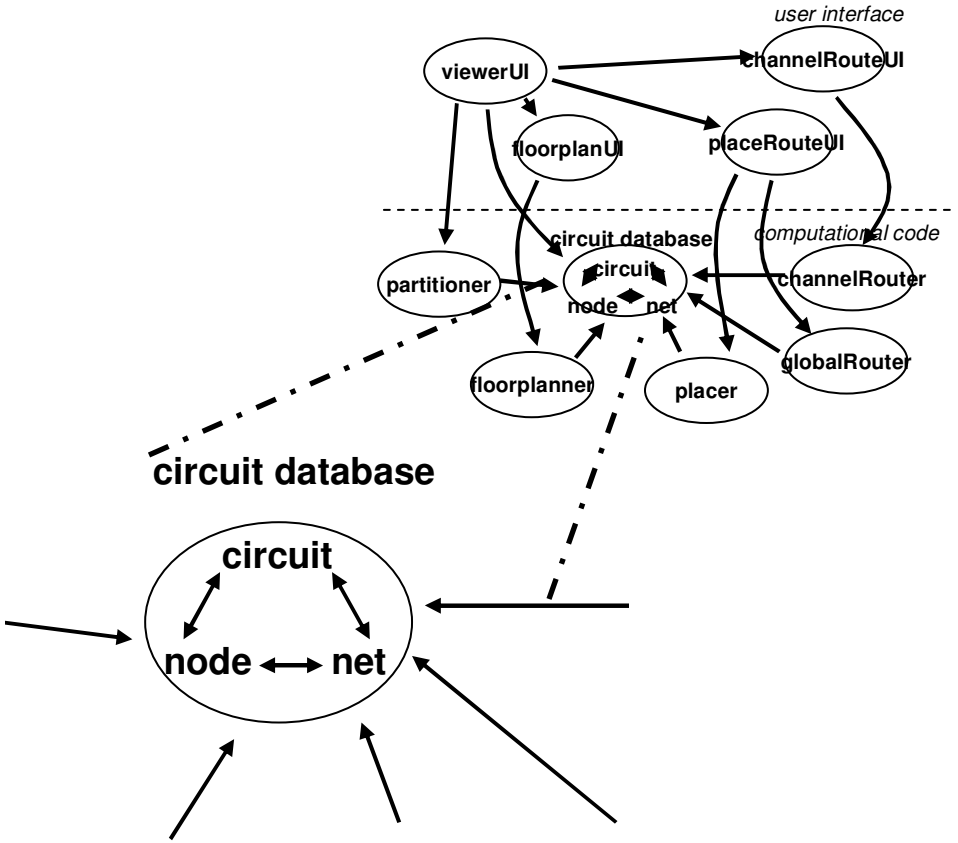
- Extracted object graph provides architectural abstraction by ownership hierarchy and by types
- Often, object graph **not isomorphic** to architect's intended architecture

# Object graph vs. target architecture

## Aphyds object graph

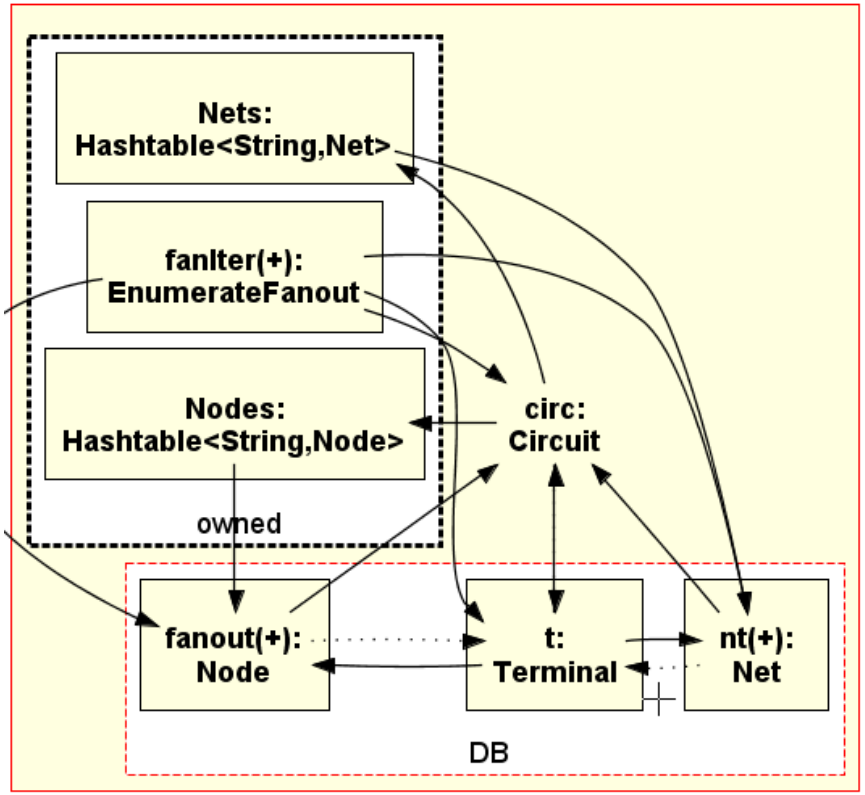



## Aphyds target architecture

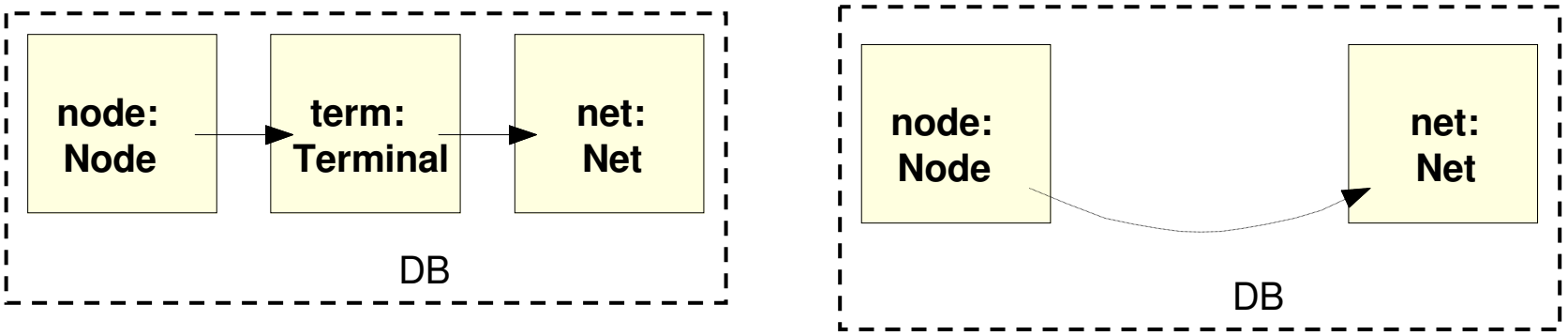


# Soundly summarize private domains

- **Private domains** hold representation
- **Public domains** hold visible state
- Eliding private domains reduces clutter
- Must be done soundly

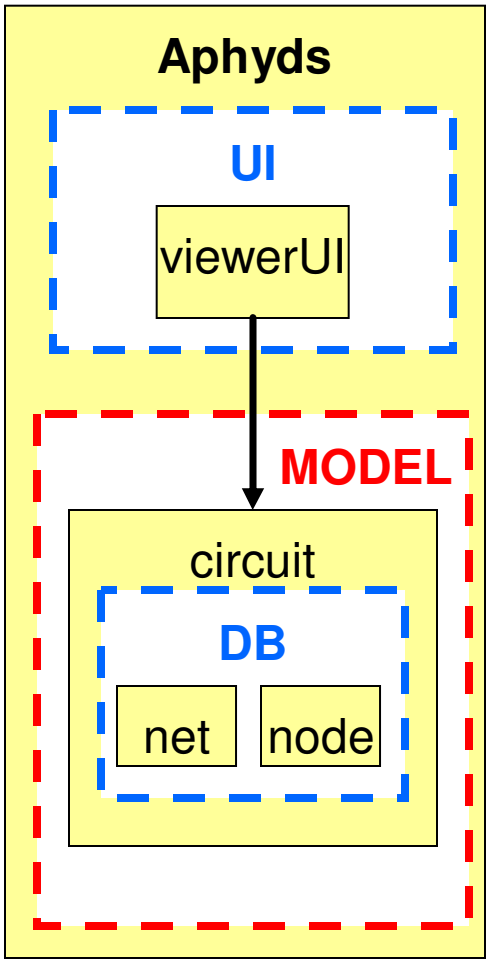


# Soundly summarizing elided objects



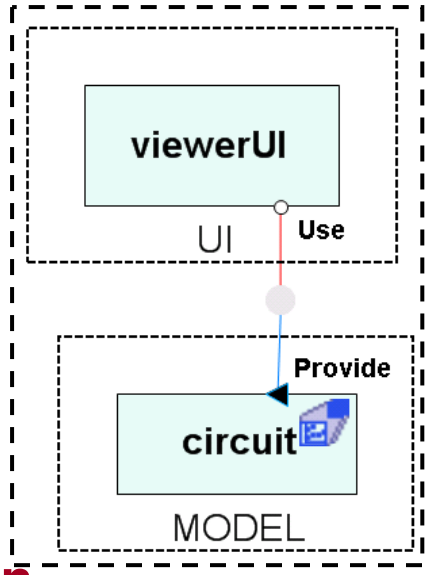

- Eliding object ‘term’ leads to **summary edge** to show **transitive communication**
- Effectively, **abstracts object into edge**
- Notion of **rich connector** in architecture

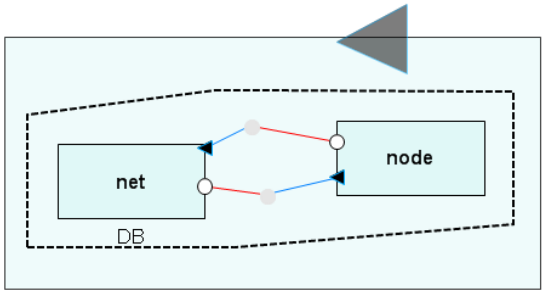
# Represent abstracted ObjectGraph as component-and-connector (C&C) view



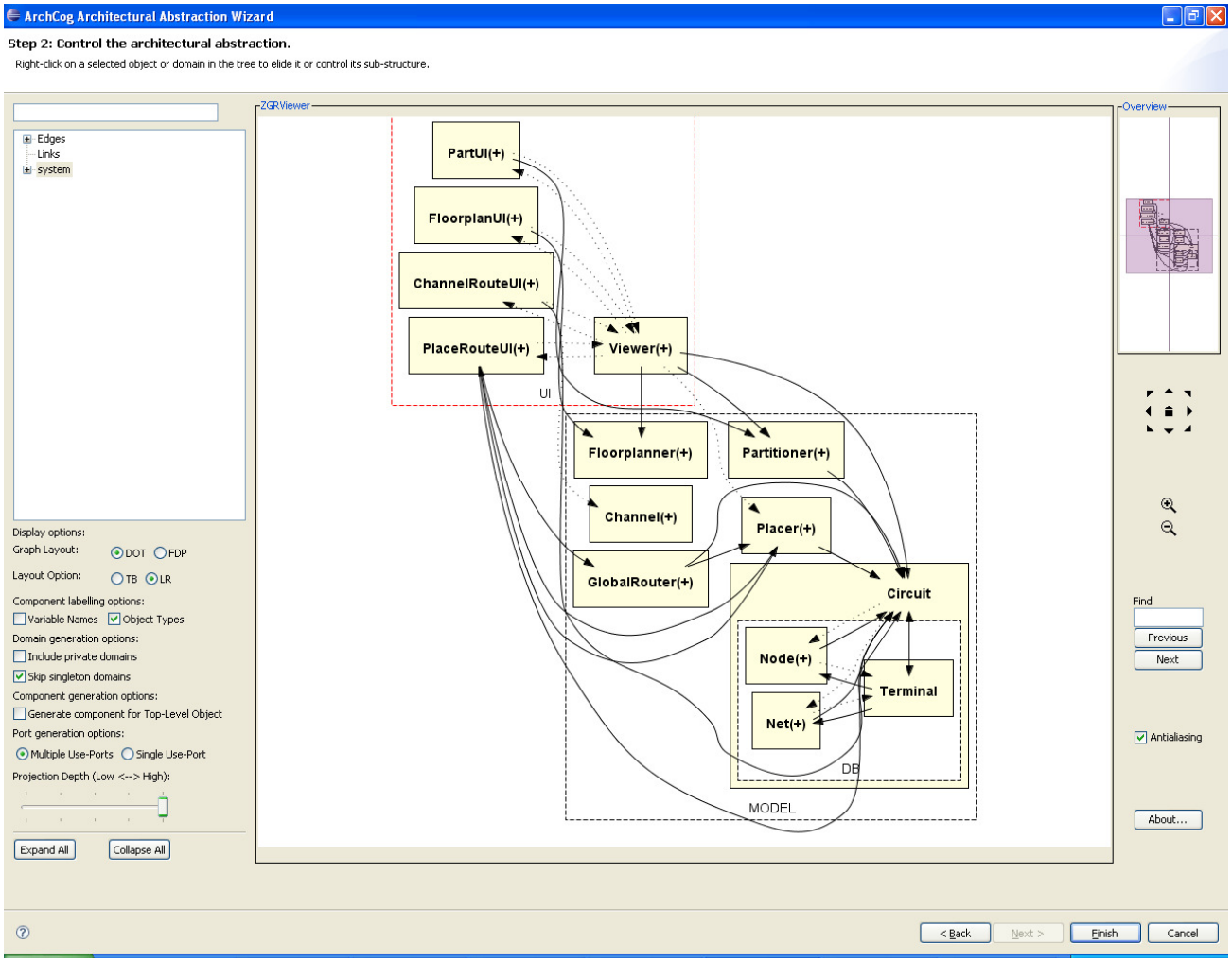

**ObjectGraph ↔ C&C view**

- Top-level object ↔ System
- Object ↔ Component
- Domain ↔ Group
- Interface ↔ **Provide** port
- Field reference ↔ **Use** port
- Object relation ↔ Connector
- Substructure ↔ **Representation**

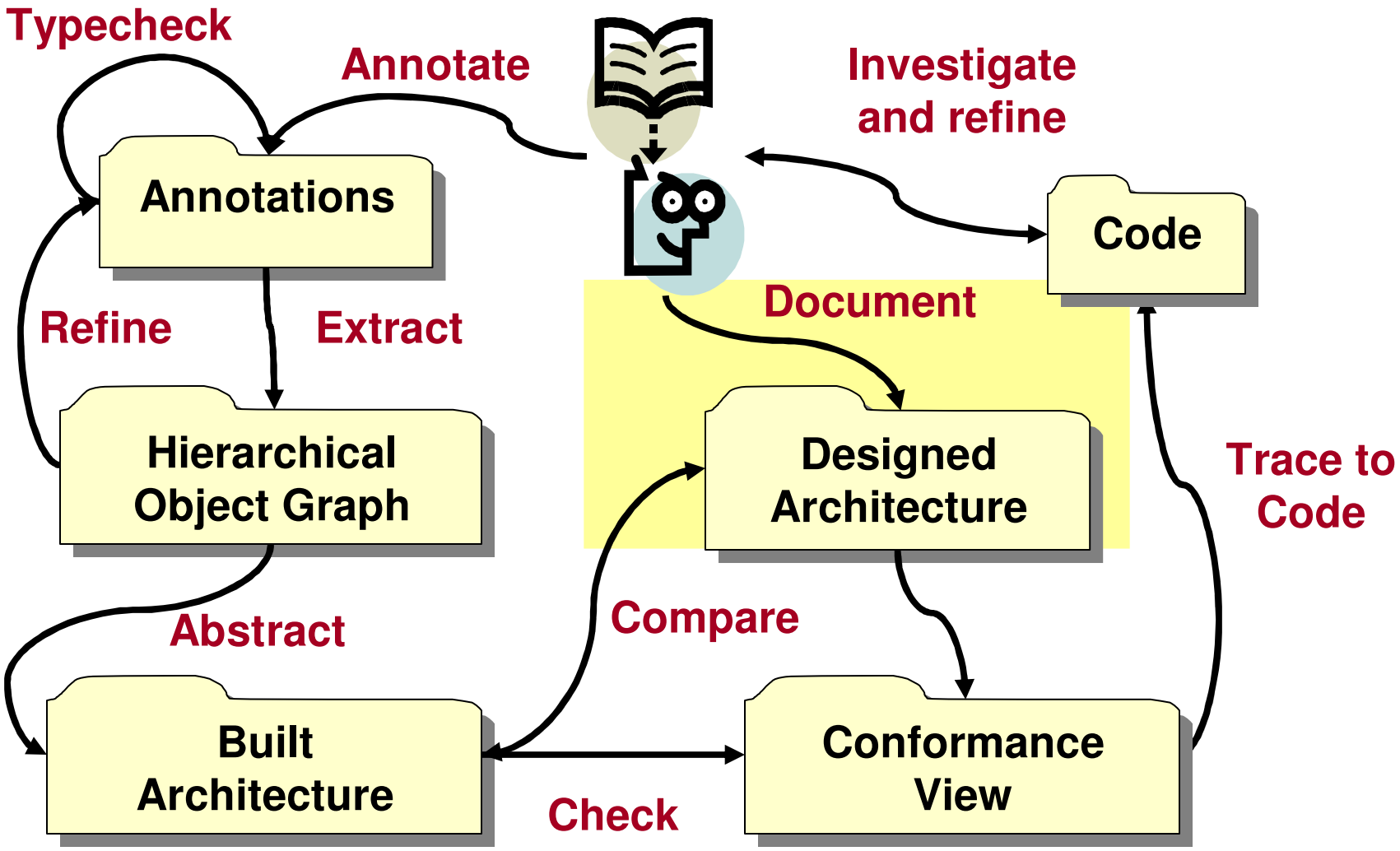


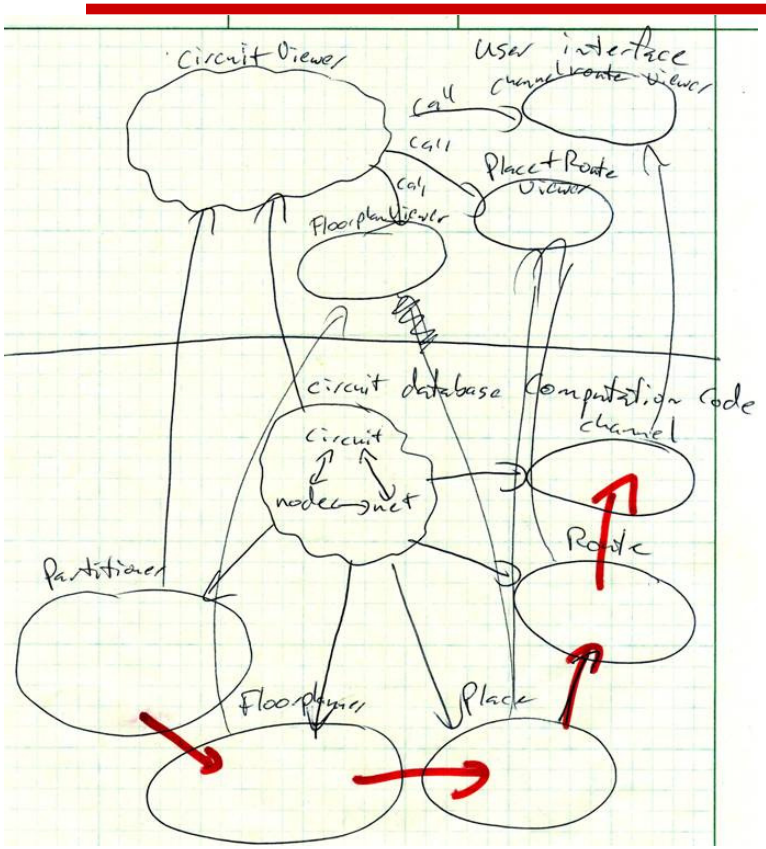
# ArchCog: abstract object graph; present in architecture description language

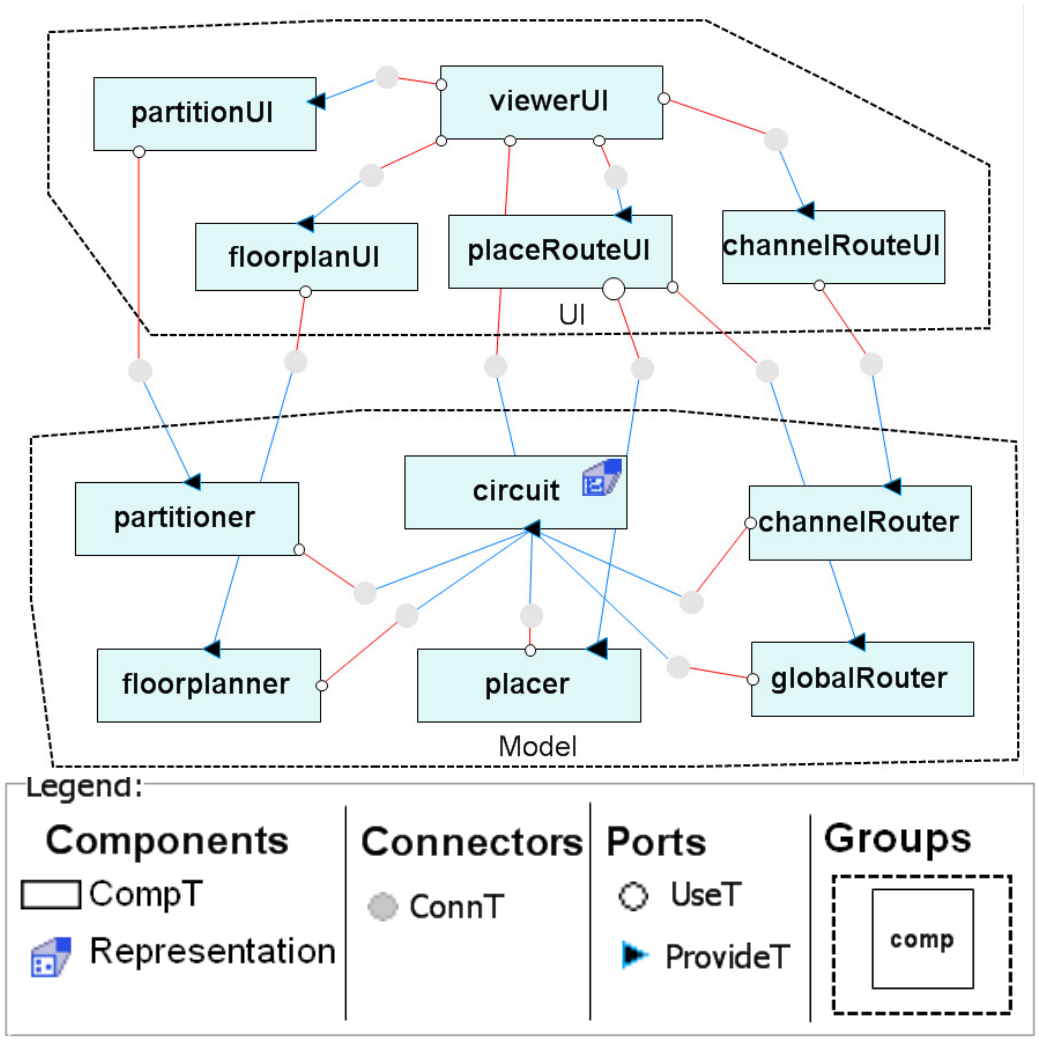


# SCHOLIA: document target architecture

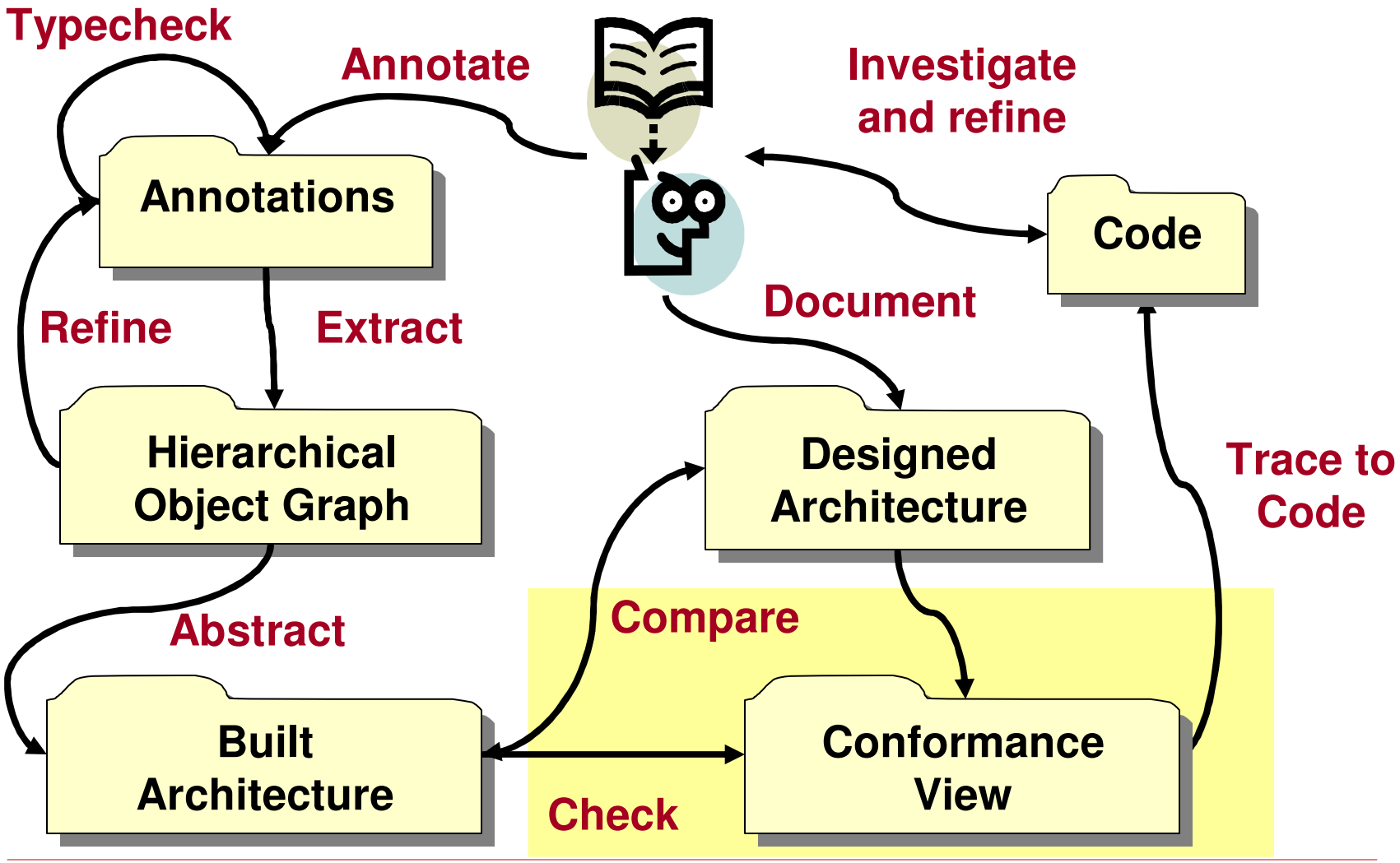


# Aphyds: document designed architecture in architecture description language



# SCHOLIA: analyze conformance



# Analyzing conformance of system to target architecture

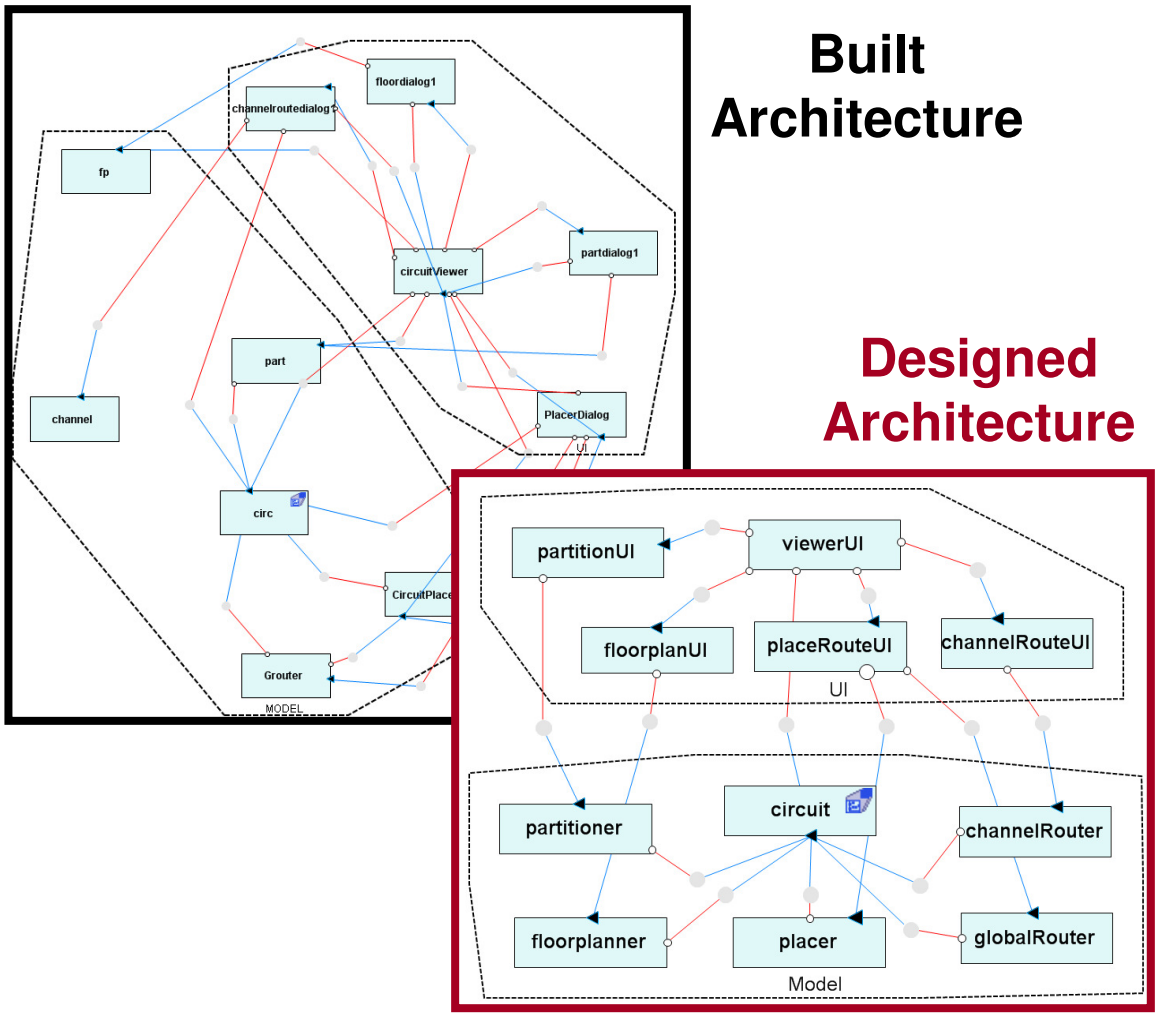
- Key property: **communication integrity**
  [Moriconi et al., TSE'95] [Luckham and Vera, TSE'95]

  *Definition: each component in the implementation may only communicate directly with the components to which it is connected in the architecture.*
- Informal diagrams **omit** communication; confirmed by experience at Microsoft
  [Murphy et al.,TSE'01] [Aldrich et al., ICSE'02]

# Why is comparing built and designed architectures hard?

- No **unique identifiers**
- **Renames**
- Insertions
- Deletions

- Solution: use structural comparison



# Structural comparison

[Abi-Antoun, Aldrich, Nahas, Schmerl and Garlan, ASE'06 and J. ASE '08]

- Exploit **hierarchy** in architectural views to match the nodes
- Detect **renames**, **insertions**, **deletions** and restricted **moves**
  - Previous architectural comparison detected only insertions and deletions
  - Lost node **properties** needed for architectural analyses
- Optionally, force/prevent matches

# Why different from view synchronization?

- View synchronization makes two architectural views **identical**
- Conformance analysis
  - Enforce **communication integrity**
  - Account for **communication in built view** that is not in designed view
  - Do not propagate all implementation objects

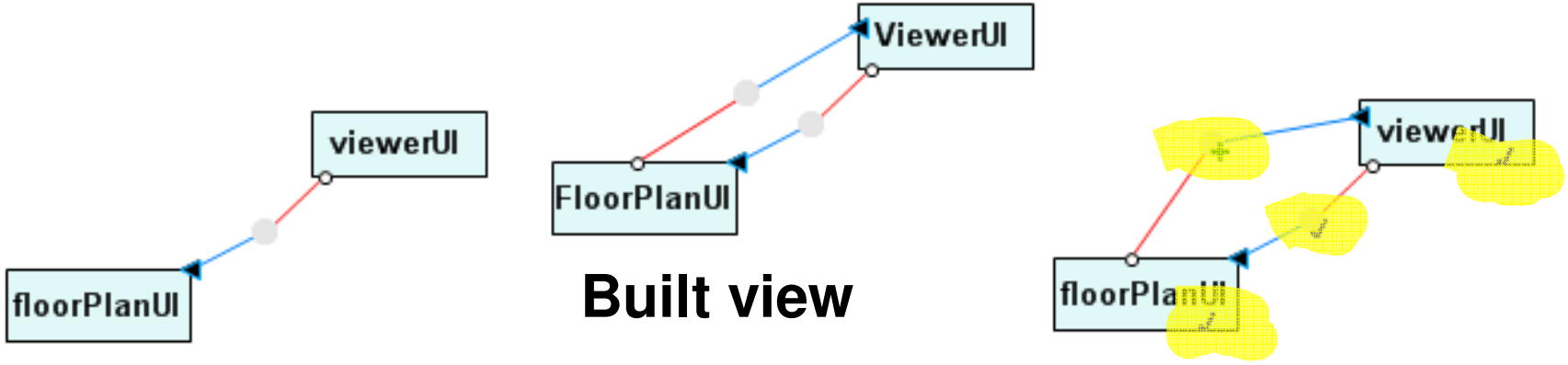
# Conformance analysis identifies following key differences

- **Convergence**: node or edge **in both** built and in designed view 
- **Divergence**: node or edge in built view, but **not in designed** view 
- **Absence**: node or edge in designed view, but **not in built** view 

Terminology adopted from Reflexion Models [Murphy et al., TSE'01]

# Highlight differing connections, but use the names from the built view

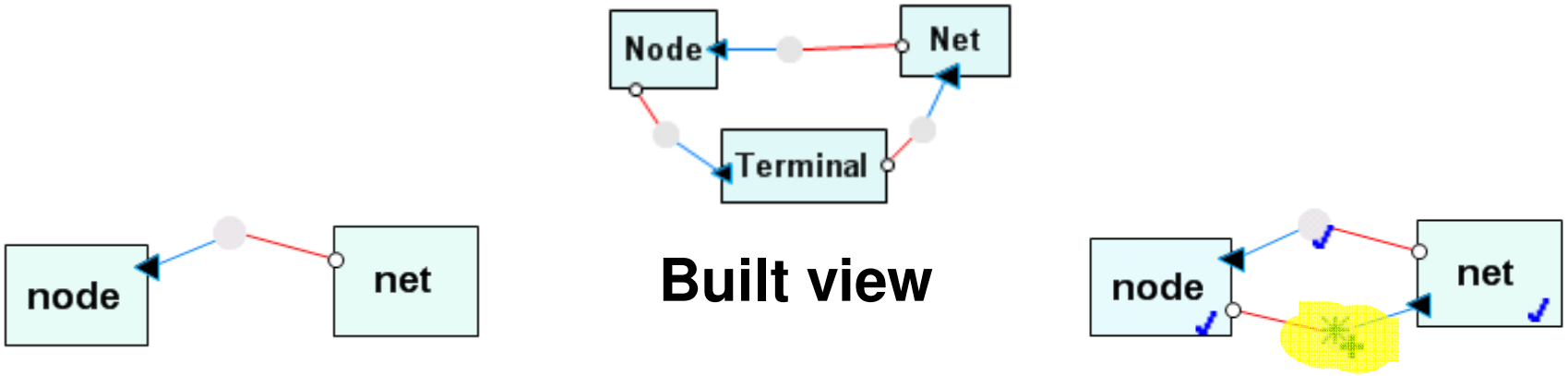
- Structurally match components in built view to those in designed view
- Show differing connections as divergences or absences




**Designed view** **Built view** **Conformance view**

# Summarize divergent components

- Do not directly propagate additional components
- Summarize additional components in built architecture using summary edges



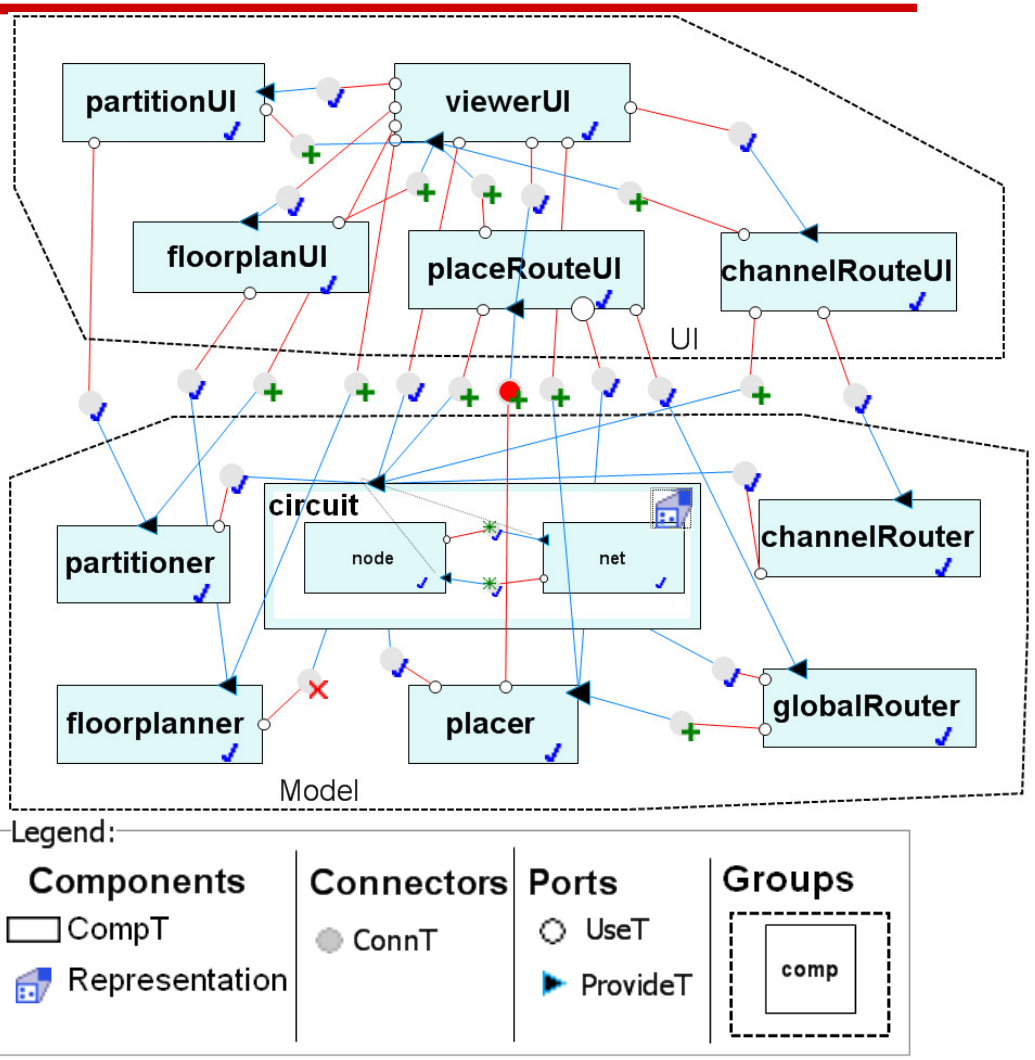

**Built view**

**Designed view**

**Conformance view**

# Developer investigates reported differences

- Study findings
- Trace to code

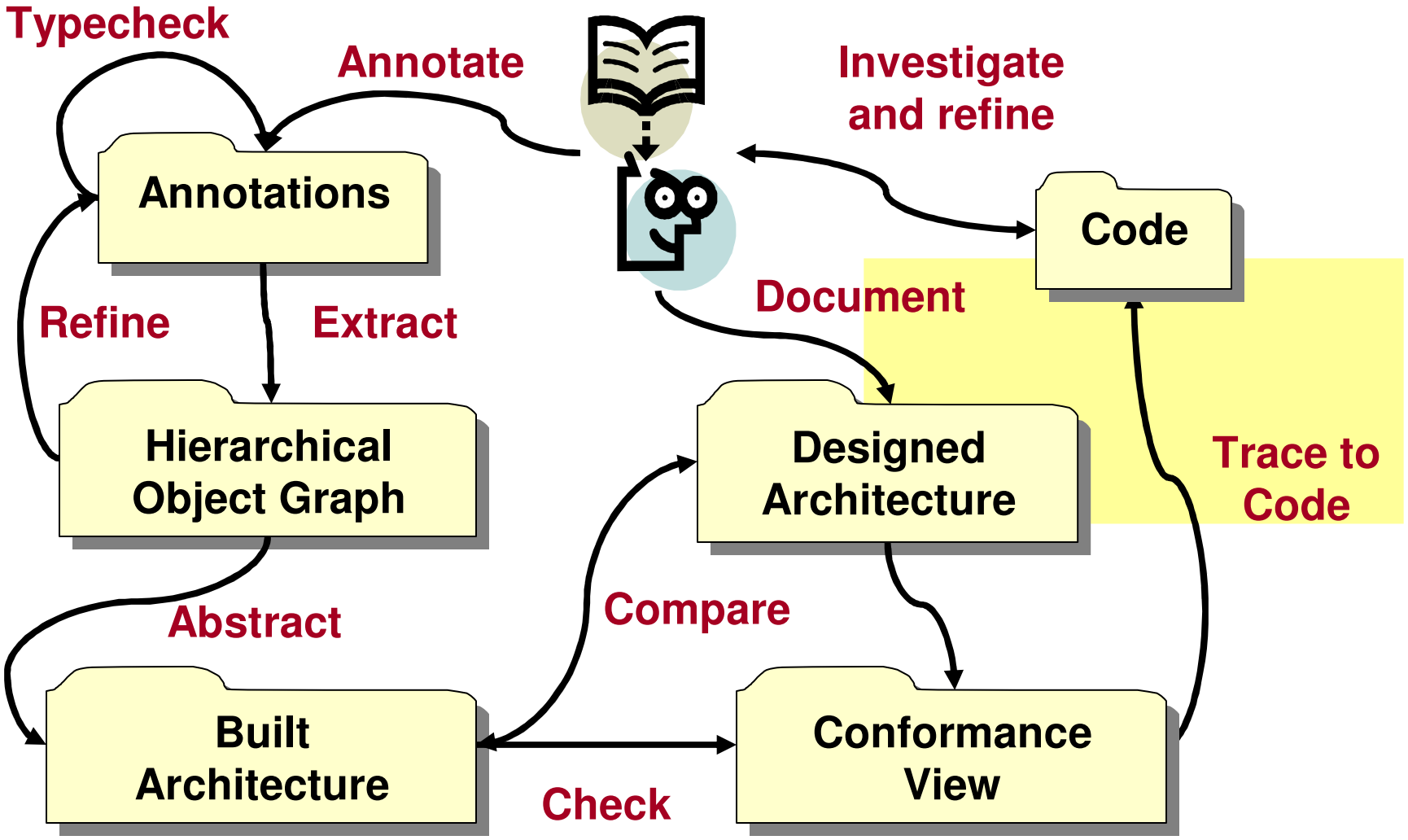



**Convergence**

**Divergence**

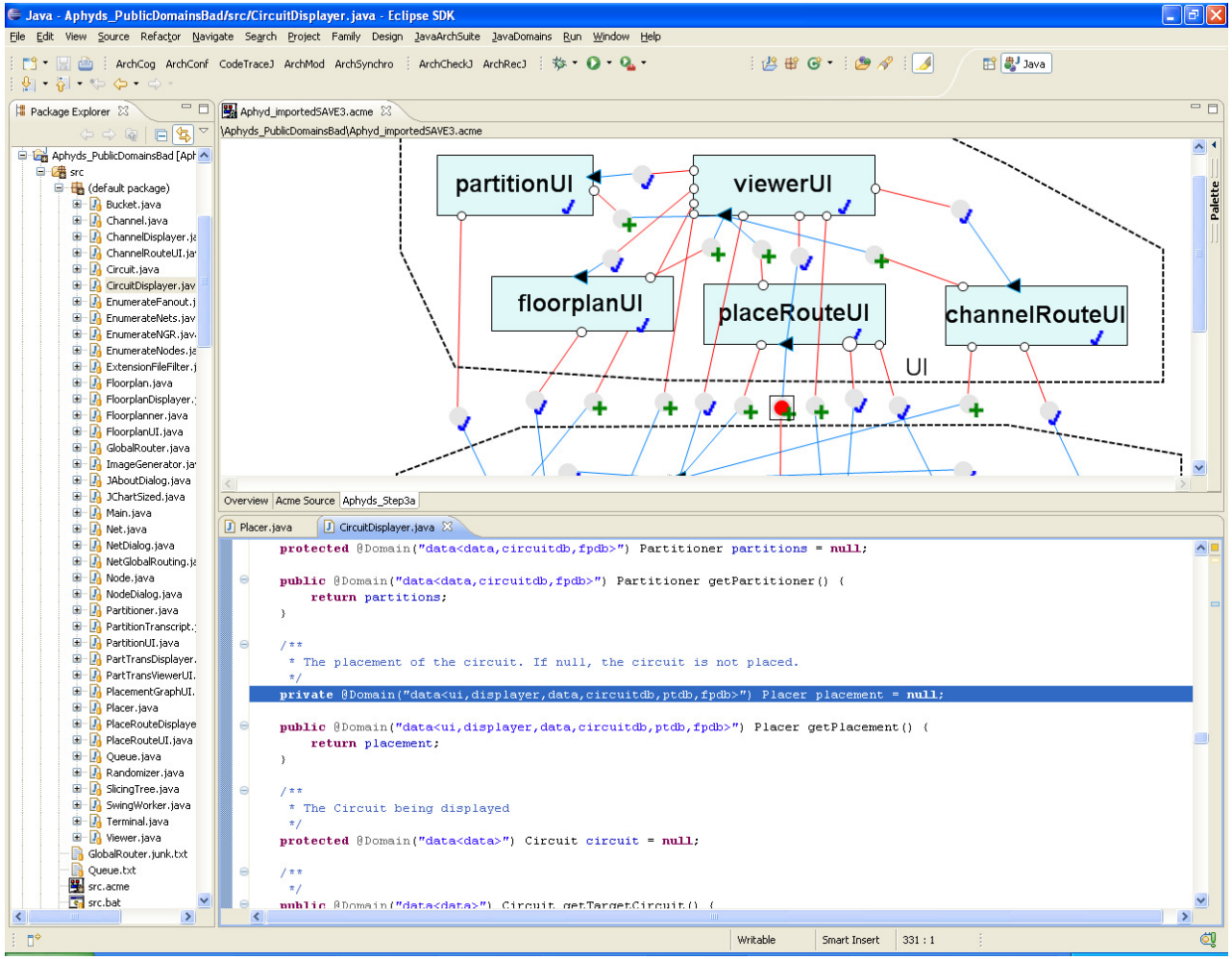
**Absence**

# SCHOLIA: trace finding to code; iterate



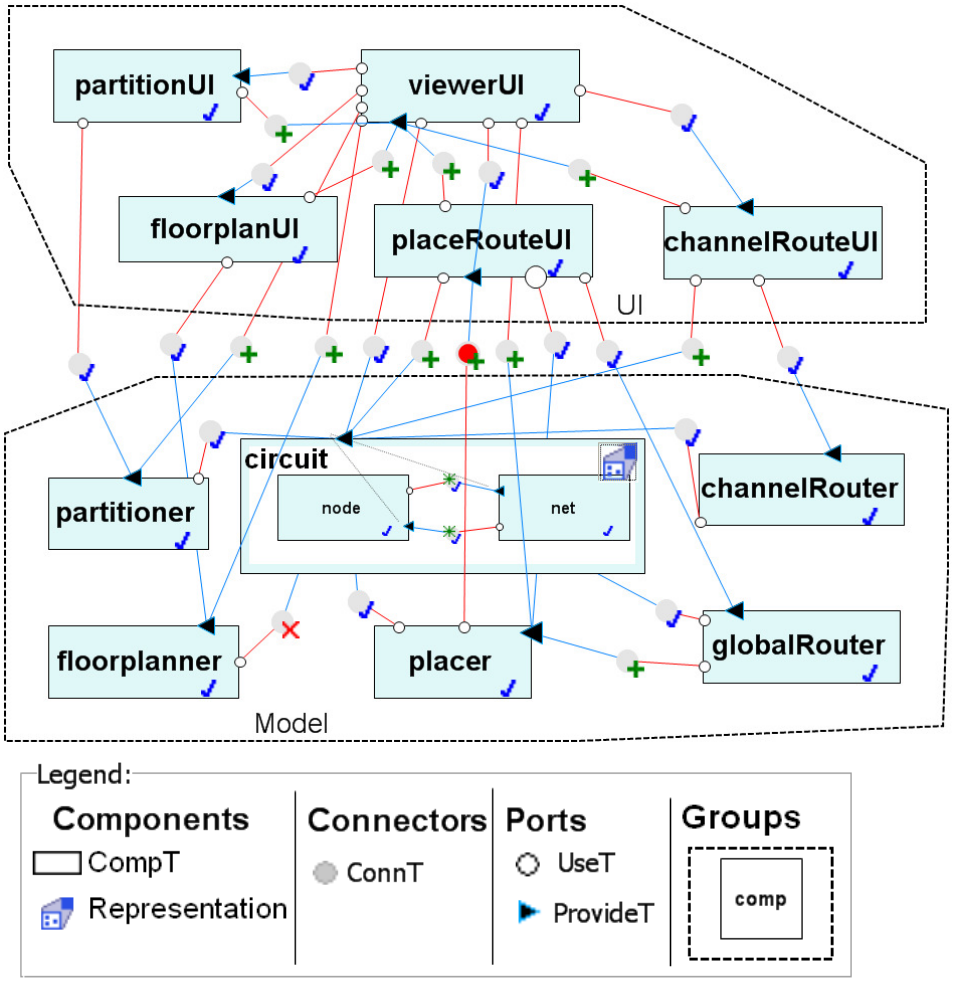
# CodeTraceJ: trace from runtime architecture to lines of code

- Trace finding to code
- Previously, only UML class diagrams supported this feature



# Aphyds: summary of findings

- **Callback** from **placer** in MODEL to **placeRouteUI** in UI (significant in a multi-threaded app)
- Many connections really **bi-directional**



# Evaluation of the SCHOLIA approach

# Several extended examples and field study

| System | Size | Comments |
|---|---|---|
| JHotDraw | 15 KLOC | Designed by experts in object-oriented analysis and design |
| HillClimber | 15 KLOC | Designed by undergraduates at UBC |
| **Aphyds** | **8 KLOC** | **Original developer drew architecture** |
| LbGrid | 30 KLOC | Extracted object graphs, showed them to outside developer |
| CryptoDB | 3 KLOC | Compelling target architecture designed by security expert |

# Limitations, Related Work and Future Work

# Limitations

- **False positives**
  - Possible in any sound static analysis
  - **Few** when developer fine-tunes annotations, controls abstraction steps, structural comparison, etc.
- Type system **expressiveness limitations**
  - Annotated systems have warnings remaining
  - Can incorporate some published research
- Manual **annotations**
  - Impractical without **annotation inference**
  - Inference active area of research

# Architectural extraction: state-of-the-art

- Extract **code architecture statically**
  - Rather than **runtime architecture**
- **Dynamic extractors**
  - [Sefika, Sane and Campbell, ICSE'96]
    [Schmerl, Aldrich, Garlan *et al.*, ICSE'04, TSE'06]
  - Runtime instrumentation
- **Static extractors**
  - Equate "component" with class, package, file
  - Class is-part-of package, etc.
  - Static extractors do not track objects precisely

# Some previous static analyses do track objects

- Object graph analyses
  - Without relying on annotations
    [Jackson and Waingold, ICSE'99,TSE'01]
    [O'Callahan, Ph.D. thesis'01] [Spiegel, Ph.D. thesis,'02]
  - Using non-ownership annotations
    [Lam and Rinard, ECOOP'03]
  - Some unsound w.r.t. aliasing or inheritance
- Points-to analysis
  - e.g., [Milanova et al., TOSEM'05]
- Shape analysis
  - e.g., [Sagiv et al., POPL'99]

# Architectural conformance: state-of-the-art

- **Static conformance** of **code architecture**
  - E.g., Reflexion Models and variants [Murphy et al., TSE'01]
- **Dynamic analysis**

  [Sefika, Sane and Campbell, ICSE'96]
  [Schmerl, Aldrich, Garlan *et al.*, ICSE'04, TSE'06]
  - Runtime instrumentation and monitoring
  - Throw runtime exception when violation occurs
  - Cannot check all possible program runs
- Conformance by design
  - **Code generation** [Shaw *et al.*, TSE'95]
  - Recent trend in model-driven development
  - Hard to use for **legacy systems**
  - More general to use **extract-abstract-check**

# Architectural conformance: state-of-the-art (continued)

- **Library-based solutions**
  [Medvidovic et al., FSE'96] [Malek, Mikic-Rakic and Medvidovic, TSE'05]
  - Relies on style guidelines
    [Luckham and Vera, TSE'95]
  - No tools to automatically enforce them
- **Language-based solutions**
  ArchJava [Aldrich et al., ECOOP'02]
  - Specify architectural constructs in code
  - Restrictions on object references
  - Require **re-engineering** existing systems
    [Aldrich, Chambers and Notkin, ICSE'02]
    [Abi-Antoun and Coelho, WICSA'05]
    [Abi-Antoun, Aldrich and Coelho, JSS'07]

# Future work

- **Increase** type system's expressiveness
- More **interactive** approach
  - Tools to add annotations
  - Interactively refine annotations, abstract object graph into C&C view, etc.
- **Quantify** benefits of runtime architectures for code modification tasks
  - Conduct **user studies** or **field studies**
  - Preliminary results from 3 pilots

# Summary

- First approach, **SCHOLIA**, to guarantee at **compile-time communication integrity** between arbitrary Java code and **hierarchical** intended **runtime architecture**
  - Uses backward-compatible statically **type-checkable annotations**
  - Instead of languages or libraries
- Evaluation on real systems very promising

# If you enjoyed this talk…

- Consider taking CSC 7110 in Winter'10
  - Learn about static analysis
    - Principles of data flow analysis
    - Write plugins using Eclipse Abstract Syntax Tree
  - Learn about dynamic analysis
    - General principles and infrastructure
    - Evaluate tool to find concurrency bugs
  - Evaluate research tools
    - E.g., SOOT, TACLE, etc.
  - Evaluate commercial tools
    - E.g., Fortify, etc.
  - Alternatively, survey literature on topic

# References


- Abi-Antoun, M. and Aldrich, J. **Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations.** *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2009.
- Abi-Antoun, M. **Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure**. Ph.D. thesis, Carnegie Mellon University, 2009. To appear as Technical Report CMU-ISR-09-119.