Software architecture: high-level description of a system’s organization

• Communication between stakeholders
• Analyzing quality attributes:
  • Maintainability,
  • Security, performance, reliability …
• Different perspectives or views:
  • Code architecture
  • Runtime architecture
  • Distinct but complementary
  • Focus today is on structure, not behavior
Code architecture shows code structure (classes, inheritance, etc.)

- **Code architecture** represents **static code structure** of system
  - Classes, packages, modules, layers, ..
  - Inherits from class, implements interface
  - Dependencies: imports, calls graphs.

- Impacts qualities like **maintainability**
- **Mature** tool support
Runtime architecture shows objects (instances) and relations between them

- **Runtime architecture** models **runtime structure** as runtime components and potential runtime interactions
  - Runtime component = sets of **objects**
  - Runtime interaction = e.g., points-to relation

- Impacts qualities such as **security**, performance, reliability, …

- **Immature** tool support
Analyze quality attribute, assuming architecture reflects all communication

- Microsoft uses **threat modeling** and claims **50% reduction** in vulnerabilities
- Security experts review hand-drawn diagrams (Vista has 1,400 diagrams)
- **Checking conformance** of implementation to architecture not addressed
- Potential security violations

Redacted diagram for Windows Vista™ subsystem
Security analysis requires runtime architecture, not code architecture
Security analysis requires **runtime architecture**, not code architecture.
Disclaimer: security architecture

- Threat modeling uses a Data Flow Diagram (DFD) with security annotations.
- This presentation uses a different architectural style: a security architecture shows points-to (not data flow) connectors, has no explicit data stores or external interactors, and uses more general boundaries that are tiers.
Example security runtime architecture

- **Runtime components** and connections
- **Hierarchical decomposition**
- **Partitioning into tiers**
Some analyses must consider worst case of possible communication

- Results valid only if model is **sound**
- **Sound**: reveal all objects and relations that may exist at runtime – in any run

Entry points in implementation not reflected in architecture
Architectural extraction’s key property: **soundness**

- **Definition**: a runtime architecture is **sound** if it represents all runtime components and all possible interactions between those components.
- Informal Visio diagram often unsound
Runtime structure distinguishes between different instances of the same class

- Different instances usually have different architectural properties
  - Here, trustLevel = Full vs. Partial
  - Usually, one java.io.File class in class diagram

[Diagram showing trust levels and relationships between classes]

Two instances can have different values for crucial trustLevel property.
Aliasing or state sharing is a challenge in representing a runtime architecture

- Impacts architectural properties
  - Settings File (trustLevel = Partial)
  - vs. Registration File (trustLevel = Full)
  - Combine these two instances into one?

Assume ‘Registration File’ and ‘Settings File’ distinct, with different values for trustLevel.

Assume one File DataStore, with one value for trustLevel.
Other key property: aliasing soundness

- **Definition**: an architecture is sound w.r.t. aliasing if no one runtime entity appears as two “components” in the architecture.
- Otherwise, could assign two different values of `trustLevel` architectural property for one true runtime entity.
Architectural extraction: state-of-the-art

• Using **static** analysis still open problem
  • Can capture **all possible** executions
  • Extract low-level **non-architectural** views
  • Analyses often unscalable

• Using **dynamic** analysis
  • Analyze one or more program runs
  • May **miss important objects or relations** that arise only **in other** program runs
  • E.g., security analysis must handle **worst**, not typical, possible runtime communication
Two components should communicate only if architecture allows them to do so

- E.g., prohibit **direct communication** between certain components, for all program runs

This edge should appear even if it occurs in 1 / 1,000,000 runs!
Checking structural conformance of system to target architecture

• Key property: communication integrity

**Definition:** each component in the implementation may only communicate directly with the components to which it is connected in the architecture.

[Moriconi et al., TSE’95] [Luckham and Vera, TSE’95]

• Informal diagrams omit communication; confirmed by experience at Microsoft

[Murphy et al., TSE’01] [Aldrich et al., ICSE’02]
Previous work to ensure conformance of runtime architecture has drawbacks

- **Runtime monitoring**
  - Cannot check all possible program runs

- **Code generation**
  - Hard to use for *existing systems*
  - More general to *extract-abstract-check*

- **Language-based solutions**
  ArchJava [Aldrich et al., ECOOP’02]
  - Restrictions on object references
  - Require re-engineering *existing systems*

- **Library-based solutions**
Today, you will learn **SCHOLIA**

**SCHOLIA:** static conformance checking of object-based structural views of architecture.

Scholia are annotations inserted on the margin of an ancient manuscript. The approach supports existing, i.e., legacy systems, and uses annotations.
First *entirely static* end-to-end approach to guarantee *communication integrity* for Java

- **SCHOLIA** relates code in widely-used object-oriented language (Java) and a *hierarchical* intended *runtime architecture*:
  - **Extract** instance structure
    - Hierarchy provides abstraction
    - Achieve *soundness*
  - **Abstract** instance structure into architecture
  - **Structurally compare** hierarchical views
  - **Check** conformance
    - Enforce *communication integrity*
At **SCHOLIA**’s core is the **static** extraction of architectural **runtime structure**

- Extract **sound object graph** that conveys **architectural abstraction** by **hierarchy** and by types
  - Uses **static analysis**
  - Achieves **soundness**
  - Relies on **backward-compatible statically type-checkable annotations**
  - Minimally invasive hints about architecture
  - Instead of using new language or library
Conformance checking uses general strategy of **extract-abstract-check**

- **Extract** instance structure
  - **Add annotations** to code
  - **Run** static analysis

- **Abstract** into **built architecture**
- **Document** **designed/target** architecture
- **Compare built** and **designed** views
- **Check** conformance
SCHOLIA conformance checking

Typecheck

Annotations

Annotate

Hierarchical Object Graph

Extract

Refine

Designed Architecture

Investigate and refine

Document

Compare

Trace to Code

Check

Conformance View

Built Architecture

Abstract

Code
<table>
<thead>
<tr>
<th>Code</th>
<th>vs.</th>
<th>Execution Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td></td>
<td>Objects</td>
</tr>
<tr>
<td>Types</td>
<td></td>
<td>Instances</td>
</tr>
<tr>
<td>Static</td>
<td></td>
<td>Runtime</td>
</tr>
</tbody>
</table>

Illustrated using example

**Aphyds**, an 8,000-line Java system
Developer posits a target hierarchical runtime architecture
Developer posits a target hierarchical runtime architecture
Class diagram shows code structure, e.g., classes, inheritance.
Object diagram shows instance structure, i.e., objects and relations.
In hierarchical object structure, an object can contain other objects
Instead of objects directly owning others, use **domains** to group related objects.

Box nesting indicates “inside”
A public domain in an object defines a conceptual group of contained objects

Box nesting indicates “inside”
Placing object ‘net’ in domain ‘DB’ inside ‘circuit’ makes ‘net’ part of ‘circuit’

Box nesting indicates “inside”
Any object that can reference ‘circuit’ can also reference ‘net’ inside ‘DB’ domain

Thin border indicates logical containment
Each object can have domains, e.g., ‘net’ has ‘OWNED’ domain and ‘terms’ inside it.

Box nesting indicates “inside”
A private domain defines a strict encapsulation or ownership model

LEGEND
- Private domain
- Public domain
- object: Type

‘terms’ is in private domain of ‘net’
‘terms’ is **strictly encapsulated inside ‘net’** and cannot be leaked/aliased to outside.
Unlike class diagram, object diagram shows distinct ‘Vector’ instances

But ‘terms’ Vectors can share other objects
Hierarchy allows varying **abstraction level**, by **collapsing or expanding objects**

(+) indicates collapsed sub-structure
Central difficulty

Architectural hierarchy not readily observable in program written in general purpose programming language
All previous static analyses extract non-hierarchical abstractions

- **Object graph analyses**
  - **Without using annotations**
    [Jackson and Waingold, ICSE’99, TSE’01]
    [O’Callahan, Ph.D. thesis’01]
  - **Using non-ownership annotations**
    [Lam and Rinard, ECOOP’03]
  - Some unsound w.r.t. aliasing or inheritance

- **Related static analyses**
  - **Points-to analysis** [e.g., Milanova et al., TOSEM’05]
  - **Shape analysis** [e.g., Sagiv et al., POPL’99]
Flat object graphs do not provide architectural abstraction

- Low-level objects mixed with architecturally significant ones
  - Show plethora of objects
  - No scale-up to large programs
- Require graph summarization to get readability [Mitchell, ECOOP’06]

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

Add **ownership annotations** and leverage them using **static analysis**
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 of hierarchy
  - Low-level objects demoted further down
Collapse objects based on ownership (and types) to achieve abstraction

Non-hierarchical graph

Hierarchical graph
Step 1

Add and check ownership domain annotations

- Annotate  - Extract  - Abstract  - Document  - Compare  - Analyze  - Investigate
SCHOLIA conformance checking

Typecheck → Annotate

Annotations

Refine → Extract

Hierarchical Object Graph

Abstract

Built Architecture

Check

Conformance View

Compare

Designed Architecture

Document

Investigate and refine

Code

Trace to Code

- Annotate
- Extract
- Abstract
- Document
- Compare
- Analyze
- Investigate
Group objects into *ownership domains*

```java
class Main {
    domain UI, MODEL;

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

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

Declarations are simplified
Domains can be declared inside each class

class Circuit {
    domain DB;
    DB Node node;
    DB Net net;
    ...
}

Declarations are simplified
Aphyds: concrete annotations

```java
@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
Circuit: private domain

@Domains(\{"owned", \})
class Circuit {
    @Domain("owned") Vector nodes;
}

- Each object has one or more domains
  - E.g., Circuit declares domains \textit{owned} and DB
- Each object is in exactly one domain
  - E.g., nodes in domain \textit{owned}
Circuit: public domain

@Domains({“owned”, “DB”})
class Circuit {
    @Domain(“owned”) Vector nodes;
    @Domain(“DB”) Node node;
}

- Each object has one or more domains
  - E.g., Circuit declares domains owned and DB
- Each object is in exactly one domain
  - E.g., nodes in domain owned; node in domain DB
Strict encapsulation vs. logical containment

(1) Strict encapsulation (private domain)

(2) Logical containment (public domain)
Example #2: Sequence

class Sequence {
    Cons head;

    public Iterator iterator() {
        return new Iterator(head);
    }
}

- Sequence has private state (head)
  - Should not be accessible to outside
- Sequence has iterators that are accessible to outside
  - Can also access private state
Sequence: private domain

@Domains({"OWNED"})

class Sequence {
    @Domain("OWNED") Cons head;

    public
    Iterator iterator() {
        return new Iterator(head);
    }
}

- Sequence has private state (head)
  - Should not be accessible to outside; in private domain
- Sequence has iterators that are accessible to outside
  - Can also access private state
Sequence: public domain

@Domains({"OWNED", "ITERS"})
class Sequence {
  @Domain("OWNED") Cons head;

  public @Domain("ITERS")
  Iterator iterator() {
    return new Iterator(head);
  }
}

- Sequence has private state (head)
  - Not accessible to outside; in private domain **OWNED**
- Sequence has iterators that are accessible to outside
  - Can also access private state; in public domain **ITERS**
Sequence: ownership domain parameters

@Domains({"OWNED","ITERS"})
@DomainParams({"ELTS"})
class Sequence {
  @Domain("OWNED<ELEMS>") Cons head;
  ...
}
@DomainParams({"ELTS"})
class Cons {
  @Domain("ELTS") Object obj;
  @Domain("OWNER<ELTS>") Cons next;
}

- To share objects across domains
- Add domain parameter to hold elements in list
Sequence object graph

- Expand the sub-structure of ‘list’
Sequence object graph (continued)

- Collapse the sub-structure of ‘list’
Strict encapsulation vs. logical containment

(1) Strict encapsulation (private domain)

(2) Logical containment (public domain)
Demo: checking Sequence annotations

• Cannot return head of Sequence
  • Head of list in **private domain**
  • **Stronger than making field private**

• Cannot nullify head of list
  • Stronger than Java visibility (e.g., **private**)

• Iterate over list
  • Iterator in **public domain ITERS**
Annotation tool support

- Use **Java 1.5 annotations**
- Typechecker uses Eclipse JDT
- Warnings in Eclipse’s **problem window**
Annotation language summary

- **@Domains**: declare domains
- **@DomainParams**: declare *formal* domain parameters
- **@DomainLinks**: declare domain link specifications
- **@DomainInherits**: specify parameters for supertypes
- **@DomainReceiver**: specify annotation on receiver
- **@Domain**: specify object annotation, *actual* domain parameters and (optionally) array parameters “annotation<domParam, …> [arrayParam, …]”
Special annotations

- **lent**: temporary alias within method
- **shared**: shared persistently or globally
- **unique**: unaliased object, e.g.,
  - newly created object
  - passed linearly from one domain to another
Annotation language

- Each object defines conceptual groups (ownership domains) for its state

@Domains: declare domains

@Domains({"owned"}) // Private domain
class Sequence {
    ...
}

- Each object is declared in a domain

@Domain: declare domain for given object

@Domain("owned") Vector v; // declare in ‘owned’ domain

Ł means instance encapsulated within object
Annotation language (continued)

• @DomainParams: declare formal domain parameters on a type
• @Domain: declare domain for object
  • Optionally specify actual domains using the parameter order in @DomainParams
• Similar to Java 1.5 generics
  • Declare formal type parameter
declare formal type parameter
class ArrayList<T> {...
  • Bind formal type parameter to actual type
ArrayList<String> seq;
Annotation language (continued)

- **@DomainInherits**: bind current type’s formal parameters to parameters of supertypes

```java
@DomainParams({"elems"})
@DomainInherits({"Iterator<elems>"})
class SeqIterator extends Iterator {
  
  }

// Again, similar to Java Generics...
class SeqIterator<T> extends Iterator<T> {
  
  }
```
**SCHOLIA**: use ArchCheckJ

- **Annotate**
- **Extract**
- **Abstract**
- **Document**
- **Compare**
- **Analyze**
- **Investigate**
SCHOLIA ArchCheckJ on Aphyds
Hands-on Exercises

- Annotate
- Extract
- Abstract
- Document
- Compare
- Analyze
- Investigate
Getting setup

- Have Java 1.5 or later installed
- Install **GraphViz**
  - graphviz-2.20.2.exe in zip file
- Read **setup.html**
- Extract zip file
  - Contains Eclipse 3.4
  - AcmeStudio 3.4.x (build 20090415N)
  - **SCHOLIA** Eclipse plugins
- **Accept license agreement**
  - CMU patent-pending technology
  - Non-commercial, research evaluation OK
Disclaimer

- **Research-Off-The-Shelf (ROTS) tools**
  - Highly specialized
  - Poorly documented
  - Mostly prototypes

- **Advice on AcmeStudio**
  - **Save early, save often (Ctrl-S)**
  - **Restart often (File → Restart)**
Exercise #1: CryptoDB

Add annotations
CryptoDB

- 3-KLOC Java
- Crypto application
Add annotations and fix warnings

- LocalKeyStore and LocalKey
LocalKeyStore and LocalKey

- Hint: use this as a guide

```
keyStore: LocalKeyStore
  owned
  keys: ArrayList<LocalKey>

localKey: LocalKey
  kekSpec: SecretKeySpec
```

KEYSTORAGE

KEYS
Exercise #1: CryptoDB

Solution
LocalKeyStore, LocalKey annotations

class LocalKeyStore {
    private domain OWNED;
    public domain KEYS;
    private OWNED List<KEYS LocalKey> keys;

    public unique List<KEYS LocalKey> getKeys() {
        unique List<KEYS LocalKey> copy = copy(keys);
        return copy;
    }
}

class LocalKey {
    private shared String keyData; // encrypted key
    private shared String keyId; // encrypted key id
    ...
}
Step 2

**Extract** hierarchical object graph using static analysis

- Annotate
- **Extract**
- Abstract
- Document
- Compare
- Analyze
- Investigate
SCHOLIA conformance checking

- Annotate
- Extract
- Abstract
- Document
- Compare
- Analyze
- Investigate
Runtime Object Graph (ROG)

- **Runtime Object Graph (ROG):** graph where
  - A node represents a runtime object,
  - An edge represents a points-to relation
Goal of **ObjectGraph** static analysis

- Extract **ObjectGraph** that **soundly approximates all possible** Runtime Object Graph (ROG)s
  - Conveys **architectural abstraction** primarily by **ownership hierarchy**
  - Optionally, merges more objects within a domain based on **their declared types**
Two phases of the static analysis

1. Build **TypeGraph**
   - Visitor over program’s Abstract Syntax Tree
   - Represents type structure of objects in code

2. Convert **TypeGraph** to **ObjectGraph**
   - **Instantiates the types** in the TypeGraph
   - Shows only objects and domains
TypeGraph: show types, domains inside types, and objects in domains
ObjectGraph: instantiate types, starting with root (user selected)
ObjectGraph: instantiate types, starting with root (user selected)
ObjectGraph: instantiate types, show domains and objects inside domains
ObjectGraph: instantiate types, show domains and objects inside domains
ObjectGraph: instantiate types, show domains and objects inside domains
ObjectGraph: instantiate types, show domains and objects inside domains
ObjectGraph: instantiate types, show domains and objects inside domains

- Annotate
- **Extract**
- Abstract
- Document
- Compare
- Analyze
- Investigate
ObjectGraph: instantiate types, show domains and objects inside domains

- Annotate  - Extract  - Abstract  - Document  - Compare  - Analyze  - Investigate
Challenge: unbounded number of objects, based on different executions

- **Invariant:** Summarize multiple objects in a domain with one canonical object

- **Invariant:** Merge two objects of the “same type” that are in the same domain
  - I.e., same declared type, or subtype thereof
  - Or of compatible types (more later)
Challenge: TypeGraph does not show all objects in each domain

- Reusable or library code often parametric with respect to ownership
- List does not “own” its elements
- Takes domain parameter $ELTS$ for elements
Challenge: TypeGraph does not show all objects in each domain

- At runtime domain parameter bound to other actual domain
- *Invariant: In the ObjectGraph, each object that is in a given domain must appear where that domain is declared*
- **Pull each object** declared inside formal domain parameter into each domain bound to the formal domain parameter
ObjectGraph: pull objects from formal domains to actual domains

class Node<OWNER> {
    domain OWNED;
    // List::ELTS Node::OWNER OWNED List<OWNER Terminal> terms;
}

class List<ELTS T> {
    // ELTS is for List elements
    // T is generic type parameter
    // Virtual field
    ELTS T obj;
}
ObjectGraph: pull objects from formal domains to actual domains

class Circuit {
    domain DB;
    // Node::OWNER  Circuit::DB
    DB Node node;
    ...
}

class Node<OWNER> {
    domain OWNED;
    OWNED List<OWNER Terminal> terms;
}

Annotate  Extract  Abstract  Document  Compare  Analyze  Investigate
Challenge: TypeGraph does not reflect possible aliasing

- **Invariant**: the same object should not appear multiple times in the **ObjectGraph**
- Ownership domain annotations give some precision about aliasing:
  - Two objects in different domains cannot alias
  - Two objects in same domain *may* alias
ObjectGraph: merge equivalent objects inside a given domain
**ObjectGraph**: add edges to represent points-to relations, incl. to pulled objects.

- Annotate • Extract • Abstract • Document • Compare • Analyze • Investigate
Challenge: ObjectGraph can have cycles

- Project **ObjectGraph** to limited depth

```
ObjectGraph
  Object
    Has-a
      Domain

root
  domain1
    object11
    domain111
      object1111
      object1112
  domain2
    object21
    object22
```
Challenge: objects from elided sub-structures could point to other objects

• Invariant: show all object relations, even ones due to elided sub-structures
• Lift edge to parent object when hidden sub-object points to external objects
Aphyds: no longer show formal domains, e.g., ‘ELTS’ inside ‘terms’
Aphyds: ‘node’ substructure points to other objects, such as ‘term’ object
Aphyds: collapsing substructure of ‘terms’ object causes edge lifting

circuit: Circuit

{term: Terminal}

node: Node

OWNED
terms (+): List<Terminal>

net: Net

DB

Object

(+) Substructure

Actual Domain

- Annotate - Extract - Abstract - Document - Compare - Analyze - Investigate
Aphyds: collapsing substructure of ‘node’ object causes additional edge lifting
Extraction key property: **soundness**

- **To be sound**, must **show** all objects and relations that may exist in any run
- **Aliasing soundness**: no one object appears as two “boxes” in object graph
Intuition behind soundness

Runtime Object Graph (ROG) ObjectGraph

- Annotate • Extract • Abstract • Document • Compare • Analyze • Investigate
SCHOLIA: use ArchRecJ

- Annotate
- Extract
- Abstract
- Document
- Compare
- Analyze
- Investigate
SCHOLIA ArchRecJ on Aphyds

- Abstract objects by ownership hierarchy
- Optionally abstract objects by types
Exercise #2: CryptoDB

Extract object graphs
Exercise #2: CryptoDB

Solution
CryptoDB OOG – no abstraction by types
CryptoDB abstraction by types

- Merge CustomerInfo and CreditCardInfo
- Both in CONSUMERS
- Merge objects when they share non-trivial least upper bound types
- User configures list of “trivial types”; by default, includes Object, Cloneable, etc.
CryptoDB OOG, with abstraction by types

- **provider(+): Provider**
- **engine(+): EngineWrapper**
- **mgr(+): CustomerManager**
- **encryptionRequest: EncryptionRequest**
- **keyStore(+): LocalKeyStore**
- **keyTool(+): KeyTool**
- **alias(+): KeyAlias**

**Classes:**
- PROVIDERS
- CONSUMERS
- KEYMANAGEMENT
- KEYSTORAGE

**Activities:**
- Annotate
- Extract
- Abstract
- Document
- Compare
- Analyze
- Investigate
Step 3

Abstract object graph into built architectures
SCHOLIA conformance checking

- Typecheck
- Annotate
- Investigate and refine
- Refine
- Extract
- Document
- Trace to Code
- Abstract
- Compare
- Check
- Code
- Designed Architecture
- Conformance View
- Built Architecture
- Hierarchical Object Graph
- Annotations

- Annotate
- Extract
- Abstract
- Document
- Compare
- Analyze
- Investigate
Why need to abstract an object graph?

- Extracted object graph provides architectural abstraction by ownership hierarchy and by types
- May not be isomorphic to architect's intended architecture
- May require further abstraction
Aphyds: object graph vs. target architecture

- Object graph
- Target architecture

- Annotate
- Extract
- Abstract
- Document
- Compare
- Analyze
- Investigate

Diagram showing relationships between components such as viewerUI, circuit, and globalRouter.
Elide and summarize domains/objects

- **Private domains** hold representation
- **Public domains** hold visible state
- Soundly summarize private domains

- Annotate • Extract • Abstract • Document • Compare • Analyze • Investigate
Soundly summarizing elided objects

- Eliding object ‘term’ leads to summary edge to show transitive communication
- Effectively, abstracts object into edge

• Annotate • Extract • Abstract • Document • Compare • Analyze • Investigate
Represent abstracted object graph in architecture description language

- **Aphyds**
  - UI
  - viewerUI
  - MODEL
  - circuit
  - DB
  - net
  - node

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

- Annotate • Extract • Abstract • Document • Compare • Analyze • Investigate
SCHOLIA: use ArchCog

- Annotate
- Extract
- Abstract
- Document
- Compare
- Analyze
- Investigate
SCHOLIA ArchCog on Aphyds

- Abstract object graph into C&C view
- Control projection depth
- Elide private domains
Exercise #3: CryptoDB

Abstract object graph
Exercise #3: CryptoDB

Solution
CryptoDB built architecture

- Annotate
- Extract
- Abstract
- Document
- Compare
- Analyze
- Investigate
Step 4

Document target architecture

- Annotate
- Extract
- Abstract
- Document
- Compare
- Analyze
- Investigate
**SCHOLIA conformance checking**

- **Typecheck**
- **Annotate**
- **Investigate and refine**

- **Annotations**
- **Hierarchical Object Graph**
- **Built Architecture**
- **Designed Architecture**
- **Document**
- **Conformance View**
- **Code**

- **Refine**
- **Extract**
- **Abstract**
- **Compare**
- **Check**

- **Annotate** • **Extract** • **Abstract** • **Document** • **Compare** • **Analyze** • **Investigate**
SCHOLIA: use AcmeStudio

- Annotate
- Extract
- Abstract
- **Document**
- Compare
- Analyze
- Investigate

- Typecheck [ArchCheckJ]
- Annotate
- Investigate and refine

- Refine [ArchRecJ]
- Extract

- Abstract [ArchCog]
- Built Runtime Architecture

- Designed Architecture
- Conformance View

- Trace to Code [CodeTraceJ]
- Check [ArchConf]
Aphyds: document designed architecture in architecture description language
Exercise #4: CryptoDB

Document target architecture
CryptoDB

- DFD Level-1

- Annotate
- Extract
- Abstract
- Document
- Compare
- Analyze
- Investigate
CryptoDB

- DFD Level-2

- Annotate • Extract • Abstract • Document • Compare • Analyze • Investigate
Exercise #4: CryptoDB

Solution
CryptoDB target architecture
CryptoDB target architecture
Step 5

**Compare** built and designed architectures
SCHOLIA conformance checking

- Annotate
- Extract
- Abstract
- Document
- Compare
- Analyze
- Investigate
Why is comparing built and designed architectures hard?

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

- Solution: use structural comparison
Structural comparison

• 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
Aphyds: comparing built and designed architectures

- Accept results of structural comparison
- Optionally, force/prevent matches
Exercise #5: CryptoDB

Compare build and target architecture

- Annotate
- Extract
- Abstract
- Document
- Compare
- Analyze
- Investigate
Exercise #5: CryptoDB

Solution
Step 6

Check conformance between built and designed architectures
**SCHOLIA conformance checking**

- **Typecheck**
- **Annotate**
- **Investigate and refine**
- **Refine**
- **Extract**
- **Document**
- **Trace to Code**
- **Abstract**
- **Compare**
- **Check**

**Flowchart:**
- Annotate → Extract → Concrete
- Investigate and refine → Document
- Trace to Code

**Nodes:**
- Annotations
- Hierarchical Object Graph
- Built Architecture
- Designed Architecture
- Conformance View

**Actions:**
- Annotate
- Extract
- Abstract
- Document
- Compare
- Analyze
- Investigate
How is conformance checking different from view differencing/merging?

• Goal is not to make the built and the designed architectures **identical**
• Account for **communication in built system** that is not in designed one
  • Do not propagate all implementation objects
  • Enforce **communication integrity**
• Measure conformance as **graph edit distance** between built and designed views
Conformance checking analysis

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

Designed view

Built view

Conformance view

- Annotate • Extract • Abstract • Document • Compare • Analyze • Investigate
Conformance check identifies key differences

- **Convergence**: node or edge in both built and in designed view ✓
- **Divergence**: node or edge in built, but **not in designed** view +
- **Absence**: node or edge in designed view, but **not in built** view ✗
**SCHOLIA:** use ArchConf

- Typecheck [ArchCheckJ]
- Annotate
- Investigate and refine
- Extract [ArchRecJ]
- Document
- Trace to Code [CodeTraceJ]
- Abstract [ArchCog]
- Compare
- Conformance View
- Refine
- Check [ArchConf]
- Annotate
- Extract
- Abstract
- Document
- Compare
- **Analyze**
- Investigate
Aphyds: developer investigates reported differences

- Study findings
- Trace to code

Convergence ✔
Divergence ✗
Absence ✗

Legend:
Components
- CompT
- Representation
Connectors
- ConnT
Ports
- UseT
- ProvideT
Groups
- comp
Exercise #6: CryptoDB

Check conformance between built and designed architectures
Exercise #6: CryptoDB

Solution
CryptoDB conformance analysis
Step 7

Investigate and trace to code

- Annotate
- Extract
- Abstract
- Document
- Compare
- Analyze
- Investigate
SCHOLIA: use CodeTraceJ

- Annotate
- Extract
- Abstract
- Document
- Compare
- Analyze
- Investigate
Aphyds: trace from runtime architecture, obtained statically, to lines of code

- Trace finding to code
- Previously, only UML class diagrams supported this feature
Aphyds: summary of findings

- **Missing** top-level component **partitionUI**
- **Callback** from **placer** in MODEL to **placeRouteUI** in UI
- Many connections really **bi-directional**
SCHOLIA conformance checking

Typecheck

Annotate

Investigate and refine

Code

Refine

Extract

Document

Trace to Code

Annotate

Abstract

Compare

Check

Basic Architecture

Hierarchical Object Graph

Designed Architecture

Conformance View

Built Architecture

Abstract

Extract

Annotate

Compare

Analyze

Investigate
Outcomes of investigating findings

a) Iteratively refine annotations based on visualizing an extracted object graph, before abstracting it;

b) Fine-tune the abstraction of an object graph into an architecture;

c) Manually guide the comparison of the built and the designed architecture, if structural comparison fails to perform the proper match;

d) Update code if she decides designed architecture is correct, but implementation violates architecture;

e) Update designed architecture if she considers implementation highlights mission in architecture
Exercise #7: CryptoDB

Investigate and trace to code
Exercise #7: CryptoDB

Solution
CryptoDB summary

• What did you learn?
Discussion
Limitations

- Manual **annotation burden**
  - Impractical without **annotation inference**
  - Active area of research

- Annotation **expressiveness limitations**

- **Extraction** does not handle
  - Distributed systems (single virtual machine)
  - Dynamic architectural reconfiguration

- **Comparison** can fail to match if views are too discrepant, quadratic in the view sizes

- **False positives possible**
  - As is the case with *any sound static analysis*
  - Few when developer fine-tunes annotations
Conclusion

- You learned about SCHOLIA, to extract statically a hierarchical runtime architecture from a program in a widely used object-oriented language, using typecheckable annotations.
- If intended architecture exists, SCHOLIA can analyze, at compile-time, communication integrity between implementation and target architecture.
- In practice, SCHOLIA can find structural differences between an existing system and its target architecture.
- SCHOLIA can establish traceability between an implementation and an intended runtime architecture.
- SCHOLIA complements architectural views of code structure or partial views of runtime architecture obtained using dynamic analysis.