Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations

Marwan Abi-Antoun   Jonathan Aldrich

School of Computer Science, Carnegie Mellon University

{marwan.abi-antoun, jonathan.aldrich}@cs.cmu.edu

Abstract

An object diagram makes explicit the object structures that are only implicit in a class diagram. An object diagram may be missing and must extracted from the code. Alternatively, an existing diagram may be inconsistent with the code, and must be analyzed for conformance with the implementation. One can generalize the *global* object diagram of a system into a runtime architecture which abstracts objects into components, represents how those components interact, and can decompose a component into a nested sub-architecture.

A static object diagram represents all objects and inter-object relations possibly created, and is recovered by static analysis of a program. Existing analyses extract static object diagrams that are non-hierarchical, do not scale, and do not provide meaningful architectural abstraction. Indeed, architectural hierarchy is not readily observable in arbitrary code. Previous approaches used breaking language extensions to specify hierarchy and instances in code, or used dynamic analyses to extract dynamic object diagrams that show objects and relations for a few program runs.

Typecheckable ownership domain annotations use existing language support for annotations and specify in code object encapsulation, logical containment and architectural tiers. These annotations enable a points-to static analysis to extract a sound global object graph that provides architectural abstraction by ownership hierarchy and by types, where architecturally significant objects appear near the top of the hierarchy and data structures are further down.

Another analysis can abstract an object graph into a built runtime architecture. Then, a third analysis can compare the built architecture to a target, analyze and measure their structural conformance, establish traceability between the two and identify interesting differences.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA 2009 October 25–29, 2009, Orlando, Florida, USA

Copyright © 2009 ACM 978-1-60558-734-9/09/10. . . $10.00

Reprinted from OOPSLA 2009, , October 25–29, 2009, Orlando, Florida, USA, pp. 1–20.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Classes and Objects

General Terms Experimentation, Languages, Theory

1. Introduction

During software evolution, the most reliable and accurate description of a software system is its source code. In addition, high-level architectural diagrams of the system's organization can be useful. For instance, a diagram can help locate the components that must be modified, or indicate the magnitude of the impact of a change based on the dependencies among entities. Often, such a diagram is missing, hence the need to extract one from the code. Alternatively, the diagram may exist but may be inconsistent with the code, hence the need to analyze its conformance with the implementation.

Reverse engineering or architectural extraction can extract various complementary high-level views. For example, a *class diagram* is an important and widely used description of an object-oriented system that shows the static *code architecture* in terms of classes and inheritance relationships. Today, many tools can recover class diagrams from code.

Another important view is an *object diagram* or *object graph*, where nodes represent objects, i.e., instances of the classes in a class diagram, and edges correspond to relations between objects. An object diagram makes explicit the structure of the objects instantiated by the program and their relations, facts that are only implicit in a class diagram. While in the class diagram a single node represents a class and summarizes the properties of all of its instances, an object diagram represents different instances as distinct nodes, with their own properties [36]. For example, Gamma et al. used a class diagram and an object diagram to explain each standard design pattern [14]. Recent empirical evidence confirms the importance of "how objects connect to each other at runtime when I want to understand code that is unknown: an object diagram is more interesting than a class diagram, as it expresses more how [the system] functions" [21].

A *static object diagram* shows all possible objects and relations between objects, across all program runs, and is recovered by static analysis over the code. A *dynamic object diagram* shows the objects and the relationships that are

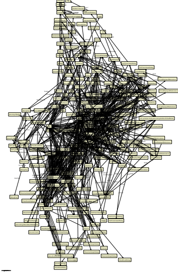


Figure 1. Aphyds object graph by WOMBLE [17]. To read the labels, zoom in by 1000%.

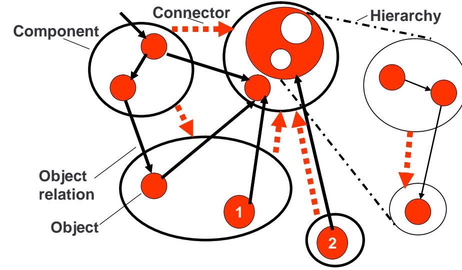


Figure 2. Architectural abstraction.

created during a specific system execution, and is recovered using a dynamic analysis [36]. Static and dynamic object diagrams provide complementary information. A static object diagram lacks precision on the actual multiplicity of the objects that the program may create, or the actual relations between objects. In contrast, a dynamic object diagram, e.g., [13], can show the exact number of instances and the actual relations in a given program run. But a dynamic object diagram may not reflect important objects or relations that show up only in other executions. For example, using a design diagram, a security review could enumerate all possible communication between trusted and untrusted parts of a system. But if the diagram under study omits communication that is present in the implementation, the analysis may be incorrect.

Scaling a flat object graph to an entire system, even a relatively small one, produces an unreadable diagram. For example, Fig. 1 is for Aphyds, an 8,000-line system. Such a diagram mixes low-level objects such as `SlicingTree` with architecturally-relevant objects from the application domain such as `GlobalRouter`, and a developer has no easy way to distinguish them.

To mitigate a diagram's complexity, hierarchy is often used to allow both high-level and detailed understanding, by expanding or collapsing selected elements [35].

Hierarchy was effective in dynamic object diagrams [16]. However, all previous static analyses extract flat static object diagrams [17, 29, 20]. Imposing hierarchy on a static object diagram is harder because architectural hierarchy is not readily observable in arbitrary code. Some language-based solutions, e.g., ArchJava [8], specify architectural hierarchy and instances directly in code. But ArchJava's breaking extensions restrict how a program uses objects and requires re-engineering an existing Java system to ArchJava [8, 5].

The proposed approach achieves hierarchy in a static object diagram by having a developer pick a top-level object as a starting point, then use local modular ownership annotations in the code [7] to impose a conceptual hierarchy on objects. Thus, architecturally significant objects appear near the top of the hierarchy and data structures further down. Similarly to ArchJava, the source code encodes the architectural intent, but instead of radically extending Java, the approach uses existing language support for annotations.

Then, a static analysis extracts from the annotated program a *global* object graph that uses object hierarchy to convey architectural abstraction. Moreover, the extracted object graph is *sound* in two respects. First, each runtime object has exactly one representative in the object graph. Second, the object graph has edges that correspond to all possible runtime points-to relations between those objects.

One can generalize the global object diagram of a system into a *runtime architecture* which abstracts one or more objects into conceptual *components*, represents how those components interact as *connectors*, and decomposes a component into a nested sub-architecture [11] (Fig. 2).

This paper proposes the SCHOLIA[1]technique to statically extract a hierarchical runtime architecture from object-oriented code, using annotations, and to analyze the conformance of an existing architecture with the code.

When the diagram is *missing*, SCHOLIA can *extract* an object graph that captures all potential executions of a program. An object graph often contains implementation details, so SCHOLIA can *abstract* it into a runtime architecture.

Alternatively, an existing diagram may be inconsistent with the code. SCHOLIA then follows the *extract-abstract-check* strategy [28], to compare and *analyze* the conformance between an extracted architecture and the intended architecture. The *communication integrity* property defines SCHOLIA's notion of conformance as: *each component in the implementation may only communicate directly with the components to which it is connected in the architecture* [22].

To our knowledge, SCHOLIA is the first approach to analyze, at compile-time, communication integrity between code in a widely-used object-oriented language and a rich, hierarchical description of the intended runtime architecture. SCHOLIA weaves several technical pieces together into an overall conformance approach.

In Section 2, we discuss the differences between the code and the runtime architecture. In Section 3, we summarize the earlier ownership type system [7]. In Section 4, we discuss the object graph extraction algorithm. In prior work [4], we described an algorithm to *extract* hierarchical object graphs from source code with ownership annotations,

[1] SCHOLIA stands for static conformance checking of object-based structural views of architecture. Scholia are annotations in a manuscript.

and proved *unique object and domain representatives*. This paper describes the extraction analysis using a clearer formalization, which allows us to prove, in addition to unique object and domain representatives, *edge soundness*. Edge soundness means that the built architecture shows all possible communication, which is a prerequisite for enforcing communication integrity.

An object graph, however, is often not isomorphic to a target architecture. The primary contribution of this paper is an integrated *extract-abstract-check* approach that *abstracts* an object graph to a standard Component-and-Connector (C&C) runtime architecture, then *analyzes* communication integrity against a target architecture. In Section 5, we discuss how SCHOLIA abstracts an object graph. In Section 6, we map an abstracted object graph into a standard component-and-connector architecture.

Also, SCHOLIA leverages our prior structural comparison algorithm [6] for the architectural comparisons in Section 7. SCHOLIA computes conformance metrics to help managers track architectural conformance over time, and derives traceability information that allows the architect to effectively trace architectural violations to code. In Section 8, we evaluate SCHOLIA and demonstrate that it can be applied to existing systems while changing only annotations in the code, that SCHOLIA can find interesting architectural violations that can be traced to code, and that SCHOLIA computes sensible conformance metrics in practice. We conclude with a discussion (Section 9) and related work in Section 10.

2. Code vs. Runtime Architecture

As a running example, we use Aphyds, a system of 8,000 source lines of Java code. A partial class diagram for Aphyds shows one `Vector` *class*, and `Node` and `Net` *classes* that have a dependency on `Vector` (Fig. 3). The class diagram suggests that a `Node` object and a `Net` object might share the same `Vector` object, but an object diagram may show this is not the case (Fig. 4(a)).

In a hierarchical object diagram, an object can contain other objects. As a result, one can collapse several nodes into one. This is a classic approach to shrink a graph. However, SCHOLIA collapses object nodes based on containment, ownership and type structures, not according to where objects are declared in the program, a naming convention or a graph clustering algorithm, as we discuss below.

Instead of objects being directly inside other objects, we use an extra level of hierarchy and group related objects inside a *domain*. A domain is similar to an architectural runtime *tier*, a *conceptual partitioning of functionality* [11].

The visualization uses box nesting to indicate containment (Fig. 4(a)). E.g., `DB` is inside `circ`. Dashed-border white-filled boxes represent domains. Sold-filled boxes represent objects. Solid edges represent field references. An object labeled `obj:T` indicates an object reference `obj` of type `T`, which we then refer to either as "object `obj`" or as "`T` object", meaning for brevity, "an instance of the `T` class".

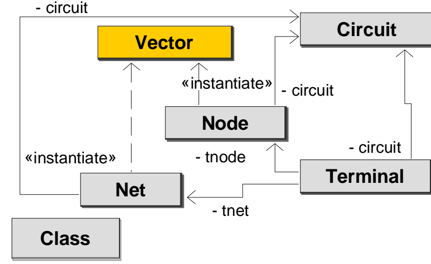


Figure 3. Code architecture of `Circuit`, `Node` and `Net`.

An object can have a *public* domain to define a conceptual group of contained objects. For instance, inside object `circ`, a public domain `DB` contains object `net`. This makes `net` *part of* `circ`. *Part of* means conceptual or logical containment, indicated by a thin border. Namely, nested objects are still accessible to the outside. For instance, an object that can reference the object `circ` can also reference the inner object `net` inside the `DB` domain.

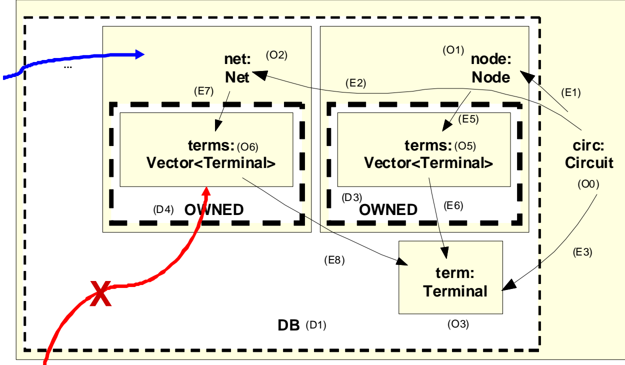
Each object can have domains. In turn, `net` has a *private domain* `OWNED` and object `terms` inside `OWNED`. A private domain defines strict instance *encapsulation* or object ownership. In other words, `terms` cannot be leaked to, nor accessed from, outside the `net` object. A thick border indicates strict encapsulation.

Unlike the class diagram which shows one `Vector` *class*, the object diagram shows distinct `Vector` objects. In turn, those two `Vector` objects refer to the same `term` object in `DB`. Finally, hierarchy allows varying the abstraction level, by collapsing or expanding the sub-structure of objects such `node` and `net`. In Fig. 4(b), the (+) symbol on an object indicates that it has a collapsed sub-structure.

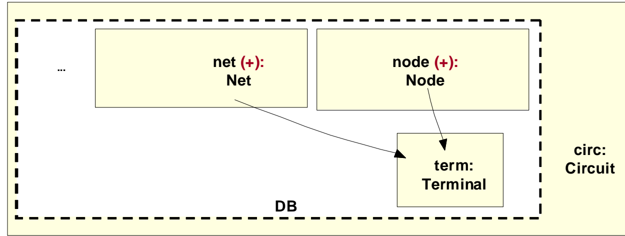
In addition, several object references that a program declares may alias, i.e., refer to the same object at runtime. An object graph such as Fig. 4(a) must conservatively show as one two objects that may alias due to subtyping, a fact that may be implicit when looking at the code. Otherwise, an architecture would be deceptive if it mapped potentially the same runtime object into two architectural components. For example, class `Stack` is a subtype of `Vector`. If there were a `Stack` object in the `OWNED` domain inside `Net`, the `Stack` and the `Vector` objects would be displayed as one.

3. Annotations

A static object diagram is extracted by a static analysis over the code. To achieve hierarchy in the object diagram, SCHOLIA relies on *local*, *modular* (one class at a time) annotations in the code that clarify the design intent. The typecheckable annotations specify object encapsulation, logical containment and architectural tiers, which are not explicit constructs in a general purpose programming language.



(a) Showing the structure of the `circ`, `net` and `node` objects.



(b) Collapsing the substructures of `net` and `node`.

Figure 4. Runtime architecture of `Circuit`, `Node` and `Net`.

```
1  class Circuit {
2    public domain DB; // Public domain
3    domain OWNED; // Private domain
4    DB Node node;
5    DB Net net;
6    DB Terminal terminal;
7    OWNED Map<String,DB Node> nodes;
8  }
9  class Node<OWNER> { // Implicit parameter
10   domain OWNED; // Private domain
11   OWNED Vector<OWNER Terminal> terms;
12 }
13 class Net<OWNER> {// Implicit parameter
14   domain OWNED; // Private domain
15   OWNED Vector<OWNER Terminal> terms;
16 }
17 class ViewerUI<M> { // Domain parameter
18   M Circuit circuit;
19 }
20 class Main { // Root class
21   domain MODEL, UI; // Top−level domains
22   MODEL Circuit circuit;
23   UI ViewerUI<MODEL> viewerUI;
24 }
```

Figure 5. Refined annotations.

The annotations assign each object to a single *ownership domain* that does not change at runtime. An ownership domain is a conceptual group of objects with an explicit name and explicit policies that govern how it can reference objects in other domains [7]. Fig. 5 shows the annotations that a developer might add to some Aphyds classes. Our tools use existing language support for annotations [2], which tends to be verbose (Fig. 6), but this paper uses a more readable syntax similar to the formal system (Fig. 9). A developer indicates the domain of an object by annotating each reference to that object in the program (lines 4–6). A developer typically chooses domain names to convey architectural intent. By convention, capital letters for domain names help distinguish them from other program identifiers.

```
1 @DomainParams({"M"}) // Domain parameter
2 class ViewerUI {
3   @Domain("M") Circuit circuit;
4 }
5 @Domains({"MODEL","UI"}) // Actual domains
6 class Main {
7   @Domain("MODEL") Circuit circuit;
8   @Domain("UI<MODEL>") ViewerUI viewerUI;
9 }
```

Figure 6. Using the concrete Java 1.5 annotations.

Each class can declare one or more *public* or *private* domains to hold its internal objects (lines 2–3), thus supporting hierarchy. Although a domain is declared at the level of a class in a program, each instance of that class has its own runtime domain. Thus, the domains within an object express a substructure within the object, one that consists of other domains and objects that represent its parts. In particular, an annotation can refer to the public domain D of an object obj, as if it were a field, using the $obj.D$ syntax. Whenever our analysis distinguishes two objects obj_1 and obj_2, it also distinguishes the domains that these objects contain in turn, such as $obj_1.D$ and $obj_2.D$.

An instance of the `ViewerUI` class accesses other objects in the `MODEL` domain, by declaring a formal *domain parameter* `M` on the `ViewerUI` class (line 17), and *binding* that parameter to domain `MODEL` (line 23). A typechecker validates the annotations and identifies where the annotations are inconsistent with each other or with the code. For instance, a `public` method cannot return an alias to an object inside a private domain. Thus, instance encapsulation is stronger than making a field be `private` to restrict its module visibility.

4. Architectural Extraction

A *Runtime Object Graph (ROG)* represents the runtime structure of an object-oriented program. Nodes correspond to runtime objects. Edges correspond to relations between objects such as *points-to* field reference relations. The goal of the static analysis is to extract from an annotated program a sound hierarchical approximation of any Runtime Object Graph, the Ownership Object Graph (OOG).

```
1   Circuit c = new Circuit();
2   OObject(c, Circuit<null>) (O0)
3   analyze(c, [])
4
5   this ↦ c, []
6   class Circuit {
7     ODomain(c.DB, Circuit::DB) (D1)
8     public domain DB;
9     ODomain(c.OWNED, Circuit::OWNED) (D2)
10    domain OWNED;
11    OObject(c.DB.nd, Node<c.DB>) (O1)
12    OEdge(c, c.DB.nd) (E1)
13    Node<DB> nd = new Node<DB>();
14    analyze(c.DB.nd, [Node::OWNER ↦ c.DB])
15    OObject(c.DB.net, Net<c.DB>) (O2)
16    analyze(c.DB.net, Net::OWNER ↦ c.DB)
17    OEdge(c, c.DB.net) (E2)
18    Net<DB> net = new Net<DB>();
19    OObject(c.DB.term, Terminal<c.DB>) (O3)
20    analyze(c.DB.term, Terminal::OWNER ↦ c.DB)
21    Terminal<DB> term = new Terminal<DB>();
22    OEdge(c, c.DB.term) (E3)
23    ...
24  }
25  this ↦ c.DB.nd, [OWNER ↦ c.DB]
26  class Node<OWNER> {
27    ODomain(c.DB.nd.OWNED, Node::OWNED) (D3)
28    domain OWNED;
29    OObject(c.DB.nd.OWNED.terms,Vector<c.DB.nd.OWNED>) (O5)
30    OWNED Vector<OWNER Terminal> terms = new Vector<...>();
31    analyze(c.DB.nd.OWNED.terms, Vector::ELTS ↦ c.DB)
32    OEdge(c.DB.nd, c.DB.nd.OWNED.terms) (E5)
33  }
34  this ↦ c.DB.nd.OWNED.terms, [ELTS ↦ c.DB]
35  class Vector<ELTS T> { T ↦ Terminal
36    lookup OObject(c.DB.term, Terminal<c.DB>)
37    OEdge(c.DB.nd.OWNED.terms, c.DB.term) (E6)
38    ELTS T obj;
39  }
```

Figure 7. Abstract interpretation of the Circuit class.

At a high level, the analysis distinguishes between objects in different domains, and abstracts objects to pairs of domains and types. The analysis also substitutes formal domain parameters with actual domains. Finally, the analysis adds edges between objects.

Object merging. Different executions may generate a different number of objects, for instance of Node objects. But a static object graph must represent all possible executions. To address this, an object graph summarizes multiple runtime objects with one canonical object in a domain, e.g., one Node object in the DB domain

Object aliasing. The object graph maintains an aliasing invariant, i.e., no one runtime object appears as two different canonical objects in the graph. The ownership domains type system give some precision about aliasing, without requiring an alias analysis. The type system guarantees that two objects in different domains cannot alias. But two objects in the same domain may alias. So, the analysis merges two objects declared *in the same domain* with the same types.

```
1   this ↦ c.DB.net, [OWNER ↦ c.DB]
2   class Net<OWNER> {
3     ODomain(c.DB.net.OWNED, Net::OWNED) (D4)
4     domain OWNED;
5     OObject(c.DB.net.OWNED.terms,Vector<c.DB.net.OWNED>) (O6)
6     OWNED Vector<OWNER Terminal> terms = new Vector<...>();
7     analyze(c.DB.net.OWNED.terms, Vector::ELTS ↦ c.DB)
8     OEdge(c.DB.net, c.DB.net.OWNED.terms) (E7)
9   }
10  this ↦ c.DB.net.OWNED.terms, [ELTS ↦ c.DB]
11  class Vector<ELTS T> { T ↦ Terminal
12    OEdge(c.DB.net.OWNED.terms, c.DB.term) (E8)
13    ELTS T obj;
14  }
```

Figure 8. Abstract interpretation of the Circuit class.

4.1 Example

The analysis takes as input a user-selected root type, in this case, Circuit (Fig. 7). First, the analysis creates an OObject (O0) for an object allocation of the root type. Then, it analyzes the class Circuit, after binding the receiver this to c.

Inside Circuit, the analysis creates an ODomain for the domain DB (D1) and another for OWNED declared in class Circuit (D2). In turn, for the object allocations inside Circuit, it creates OObjects nd (O1), net (O2) and term (O3) inside DB, and an OObject nodes inside OWNED (O4). Then, the analysis adds OEdges from O0 to O1 (E1), O0 to O2 (E2) and O0 to O3 (E3).

The analysis then processes class Node by binding the receiver to c.DB.nd. In Node, the analysis creates an ODomain for OWNED (D3), and an OObject for terms (O5). In addition, the analysis adds an OEdge (E5) from O1 to O5.

Next, the analysis processes terms by binding the receiver to c.DB.OWNED.terms, and interpreting the virtual field declaration obj inside Vector. After substituting formals to actuals, the analysis finds all OObjects in the ODomain c.DB, the types of which are subtypes of Terminal. For instance, the analysis finds the OObject O3. So, it creates an OEdge (E6) from the OObject corresponding to the terms (O5), to that OObject (O3). Similarly, the

$$
\begin{array}{rcl}
cdef & ::= & \texttt{class}\ C\texttt{<}\overline{\alpha},\overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>} \\
 & & \{\ \overline{dom}\ \ \overline{T\ f}\ \ \overline{md}\ \} \\
dom & ::= & [\texttt{public}]\ \texttt{domain}\ d; \\
md & ::= & T_R\ m(\overline{T}\ \overline{x})\ T_{\texttt{this}}\ \{\ \texttt{return}\ e_R;\ \} \\
e & ::= & x \mid \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}() \mid e.f \mid e.m(\overline{e}) \mid \ell \mid \ell \triangleright e \\
n & ::= & d \mid v \\
p & ::= & \alpha \mid n.d \mid \texttt{shared} \\
T & ::= & C\texttt{<}\overline{p}\texttt{>} \\
v, \ell & \in & locations \\
S & ::= & \ell \mapsto C\texttt{<}\overline{p}\texttt{>}(\overline{v}) \\
\Sigma & ::= & \ell \mapsto T \\
\Gamma & ::= & x \mapsto T
\end{array}
$$

Figure 9. Simplified FDJ abstract syntax [7].

$$
\begin{array}{rcll}
G \in \mathsf{OGraph} & ::= & \langle \textbf{Objs} = PtO, \textbf{Doms} = PtD, \textbf{Edgs} = PtE \rangle & \\
 & ::= & \langle PtO, PtD, PtE \rangle & \\
D \in \mathsf{ODomain} & ::= & \langle\ \textbf{Id} = D_{id}, \textbf{Domain} =\ C{::}d\ \rangle & \\
 & ::= & \langle\ D_{id},\ C{::}d\ \rangle & \\
O \in \mathsf{OObject} & ::= & \langle\ \textbf{Id} = O_{id},\ \textbf{Type} = C\texttt{<}\overline{D}\texttt{>}\ \rangle & \\
 & ::= & \langle\ O_{id}, C\texttt{<}\overline{D}\texttt{>}\ \rangle & \\
E \in \mathsf{OEdge} & ::= & \langle\ \textbf{From} = O_{src}, \textbf{Field} = f,\ \textbf{To} = O_{dst}\ \rangle & \\
 & ::= & \langle\ O_{src}, f,\ O_{dst}\ \rangle & \\
PtO & ::= & \emptyset \mid PtO \cup \{\ O\ \} & \text{Object map} \\
PtD & ::= & \emptyset \mid PtD \cup \{\ (O,d) \mapsto D\ \} & \text{Domain map} \\
PtE & ::= & \emptyset \mid PtE \cup \{\ E\ \} & \text{Edge map} \\
\Upsilon & ::= & \emptyset \mid \Upsilon \cup \{\ C\texttt{<}\overline{D}\texttt{>}\ \} & \text{Visited objects} \\
H & ::= & \ell \mapsto O & \text{Runtime object map} \\
K & ::= & \ell.d \mapsto D & \text{Runtime domain map} \\
D_{shared} & ::= & \langle\ D_s,\ ::\texttt{shared}\ \rangle & \text{Shared domain} \\
O_{world} & ::= & \langle\ O_{world},\ \texttt{Object<>}\ \rangle & \text{Root context}
\end{array}
$$

Figure 10. Data type declarations for the OGraph.

analysis processes `net` and `terms` inside `Net` (Fig. 8). Note how the "domain sensitivity" of the analysis allows it to map the same virtual field declaration (lines 35, 50) to two different OEdges in the OGraph, E6 and E8, respectively.

4.2 Formalization

Syntax. We formalize the analysis following ownership domains and Featherweight Domain Java (FDJ) [7]. We simplified the FDJ abstract syntax (Fig. 9) to exclude generic types, casts, etc. In FDJ, a type $C\texttt{<}\overline{d}\texttt{>}$ consists of the class of an object and actual ownership domain parameters. An overbar represents a sequence. The first actual domain, d_1, is the owner (*Aux-Owner* [7]). In FDJ, locations represent object identity. A store S maps a location ℓ to its contents, the type of the object, and the values stored in its fields. $S[\ell]$ denotes the store entry for ℓ. Each $\ell.d$ refers to a domain named d that is part of the runtime object ℓ. $S[\ell, i]$ denotes the value in the ith field of $S[\ell]$. The store type Σ gives a type to each location in S, one that is consistent with the classes and actual ownership domain parameters in S. $dom()$ returns the mathematical domain of a mapping, $rng()$ its range.

Data Types. In Fig. 10, an OGraph G is the triplet $G = \langle PtO, PtD, PtE \rangle$. PtO is a set of OObjects. PtD maps a pair consisting of an OObject O and a local domain or a domain parameter d in the abstract syntax, i.e., (O, d), to an ODomain D. Effectively, PtD maintains a mapping from formal domain parameters to actual domains. PtE is a set of OEdges.

The analysis distinguishes between different instances of the same class C that are in different domains, even if created at the same `new` expression. In addition, the analysis treats an instance of class C with actual parameters $\overline{p}$ differently from another instance that has actual parameters $\overline{p'}$. Hence, the datatype of an OObject uses $C\texttt{<}\overline{D}\texttt{>}$ instead of just a type and an owning ODomain. As in FDJ, an OObject's owning ODomain is the first element D_1 of $\overline{D}$.

A domain d is declared at the level of a class C in a program, but each instance of class C gets its own runtime domain $\ell.d$. Whenever the analysis distinguishes two runtime objects ℓ and ℓ', it also distinguishes the domains that these objects contain in turn, such as $\ell.d$ and $\ell'.d$. Because each runtime domain $\ell.d$ has an ODomain representative, a domain declaration d in the code can create multiple ODomains D_i.

To deal with recursive types, as we discuss later, an ODomain can have multiple parent OObjects, rather than a single one, so an ODomain does not have an owning OObject in its representation.

Each OEdge E is a directed edge from a source OObject to a target OObject, and indicates the field named `f`.

Abstract Interpretation. The analysis is an abstract interpretation of the program that maps concrete domain and field declarations in the program to abstract values in an OGraph, namely OObjects, ODomains, and OEdges.

We use a constraint-based specification (Fig. 11) instead of transfer functions, which makes it easier to prove soundness. The judgement form is as follows:

$$\Gamma, \Upsilon, PtO, PtD, PtE \vdash_{O,\ H} e$$

Γ is the typing context (Fig. 9). Υ is needed for handling recursion (Fig. 10), as we discuss later. The O subscript on the turnstile captures the context-sensitivity. H is part of the instrumentation that maps locations to OObjects (Fig. 10). We omit H for most of the rules that do not need it.

The interpretation starts with a program P consisting of a class table CT and a root expression e, which gets analyzed in the context of O_{world}. We require an OObject, O_{world}, which has a single ODomain, D_{shared}, which corresponds to the global domain `shared`. For clarity, we quality a domain d by the class that declares it, as $C{::}d$. We qualify `shared` as `::shared`.

$$\frac{\begin{array}{c}\forall i \in 1..|\overline{p}| \qquad D_i = PtD[(O, p_i)] \qquad params(C) = \overline{\alpha} \\ O_C = \langle\, O_{id},\ C\texttt{<}\overline{D}\texttt{>}\,\rangle \qquad \{O_C\} \subseteq PtO \qquad \{(O_C, \alpha_i) \mapsto D_i\} \subseteq PtD \\ PtO, PtD, PtE \vdash_O ptdomains(C\texttt{<}\overline{p}\texttt{>}, O_C) \\ PtO, PtD, PtE \vdash_O ptfields(C\texttt{<}\overline{p}\texttt{>}, O_C) \\ \forall m.\ mbody(m, C\texttt{<}\overline{p}\texttt{>}) = (\overline{x} : \overline{T},\ e_R) \\ C\texttt{<}\overline{D}\texttt{>} \notin \Upsilon \Longrightarrow \{\overline{x} : \overline{T},\ \texttt{this} : C\texttt{<}\overline{p}\texttt{>}\}, \Upsilon \cup \{C\texttt{<}\overline{D}\texttt{>}\}, PtO, PtD, PtE \vdash_{O_C} e_R\end{array}}{\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}()\overline{e}}\ [\text{PT-NEW}]$$

$$\frac{\forall(\texttt{domain}\ d_j) \in domains(C\texttt{<}\overline{p}\texttt{>}) \qquad D_j = \langle D_{id_j},\ d_j\,\rangle \qquad \{(O_C, d_j) \mapsto D_j\} \subseteq PtD}{PtO, PtD, PtE \vdash_O ptdomains(C\texttt{<}\overline{p}\texttt{>}, O_C)}\ [\text{PT-DOM}]$$

$$\frac{\begin{array}{c}\forall(T_k\ f_k) \in fields(C\texttt{<}\overline{p}\texttt{>}) \qquad owner(T_k) = p'_k \qquad D_k = PtD[(O, p'_k)] \\ \forall O_k\ PtO, PtD, PtE \vdash_O ptlookup(D_k, T_k) = O_k \quad \{\langle\, O_C,\ f_k,\ O_k\,\rangle\} \subseteq PtE\end{array}}{PtO, PtD, PtE \vdash_O ptfields(C\texttt{<}\overline{p}\texttt{>}, O_C)}\ [\text{PT-FIELDS}]$$

$$\frac{\begin{array}{c}O_k = \langle\, O_{id}, C\texttt{<}\overline{D}\texttt{>}\,\rangle \in PtO \qquad D_1 = D \qquad T' = C'\texttt{<}\overline{p'}\texttt{>} \qquad C <: C' \\ \forall i \in 1..|\overline{p'}| \qquad D'_i = PtD[(O, p'_i)] \qquad D'_i = D_i\end{array}}{PtO, PtD, PtE \vdash_O ptlookup(D, T') = O_k}\ [\text{PT-LOOKUP}] \qquad \frac{}{\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O x}\ [\text{PT-VAR}]$$

$$\frac{}{\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O \ell}\ [\text{PT-LOC}] \qquad \frac{\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O e_0}{\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O e_0.f_k}\ [\text{PT-READ}]$$

$$\frac{\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O e_0 \qquad \Gamma, \Upsilon, PtO, PtD, PtE \vdash_O \overline{e}}{\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O e_0.m(\overline{e})}\ [\text{PT-INVK}]$$

$$\frac{O_C = H[\ell] \qquad \Gamma, \Upsilon, PtO, PtD, PtE \vdash_{O_C} e}{\Gamma, \Upsilon, PtO, PtD, PtE \vdash_{O,\ H} \ell \rhd e}\ [\text{PT-CONTEXT}]$$

$$\frac{\begin{array}{c}\forall \ell \in dom(S), \Sigma[\ell] = C\texttt{<}\overline{p}\texttt{>} \qquad H[\ell] = O = \langle O_{id}, C\texttt{<}\overline{D}\texttt{>}\rangle \in PtO \\ \forall m.\ mbody(m, C\texttt{<}\overline{p}\texttt{>}) = (\overline{x} : \overline{T},\ e_R) \qquad \{\overline{x} : \overline{T},\ \texttt{this} : C\texttt{<}\overline{p}\texttt{>}\}, \emptyset, PtO, PtD, PtE \vdash_{O,\ H} e_R\end{array}}{PtO, PtD, PtE \vdash_{CT,H} \Sigma}\ [\text{PT-SIGMA}]$$

Figure 11. Constraint-based specification of the object graph extraction analysis.

In PT-NEW, the analysis interprets a `new` object allocation in the context of an OObject O as follows. First, PT-NEW checks that PtO has an OObject O_C for the newly allocated object. Since PtD maintains the binding from each formal domain parameter to some ODomain, PT-NEW ensures that the representatives of the actual domains $\overline{p}$ bound to the parameters of class C are in PtD.

PT-NEW then uses the auxiliary judgement PT-DOM to ensure that PtD has an ODomain corresponding to each domain that the class C locally declares. In PT-DOM, the $domains$ auxiliary judgement from FDJ returns the ownership domains that a class declares, after substituting formal domain parameters with actual domains. $domains$ also includes inherited domains, including the private domains. In FDJ, *private* domains are misnamed, and really have a *protected* semantics [7, Rule *Aux-Domains* (Fig. 14)].

PT-NEW then relies on the auxiliary judgement PT-FIELDS to ensure that PtE has an OEdge from O_C to each object in the target domain that is type compatible with the target type, using PT-LOOKUP. In PT-FIELDS, the $fields$ auxiliary judgement from FDJ returns the fields that a type declares, after substituting formal domain parameters with actual domains, and includes inherited fields.

Finally, PT-NEW obtains each expression e' in each method m in C, and processes e' in the context of the OObject O_C. Before PT-NEW checks these expressions recursively, it adds the current combination of a type and actual domain parameters to Υ. If PT-NEW discovers by looking at Υ that it previously analyzed that same combination, it does not recurse into the same OObject, thus avoiding infinite recursion.

Although the case for new expressions is the most interesting, the analysis requires rules for all the expression types to make the induction work. The rules for PT-VAR, PT-LOC, PT-READ, and PT-INVK are self explanatory.

PT-CONTEXT analyzes method calls in progress $\ell \rhd e$, where ℓ is the receiver, by moving into the context of the receiver object O_C. Finally, the induction requires an aug-

mented store typing rule, PT-SIGMA, to ensure that method bodies have been analyzed for all objects in the store.

Recursion. The analysis must handle recursive types, which can lead an OGraph to grow arbitrarily deep. To ensure termination, the OGraph is finite, and can contain cycles. The analysis creates a cycle in the OGraph when it reaches a "similar" context. We chose to unify domains. For instance, in Fig. 12, the `OWNED` domain inside `nwQT` is the same as the `OWNED` domain inside `aQT`. Because the same ODomain can now appear as the child of two OObjects, an ODomain cannot have an owning OObject. The visualization, however, expands the OGraph to a limited depth—the user sees the graph above the thick dashed line Fig. 12.

4.3 Soundness

The soundness proof relies on an instrumentation of the FDJ runtime semantics, an approximation relation, and standard Progress and Preservation theorems. We summarize below the key results. In addition to the FDJ store S, the instrumentation maintains the maps H and K (Fig. 10).

The instrumented evaluation has the judgement form: $e; S; H; K \rightsquigarrow_G e'; S'; H'; K'$ where $G = \langle PtO, PtD, PtE \rangle$ is the statically computed object graph.

This instrumentation is safe since discarding it produces exactly the previous semantics. In IR-NEW (Fig. 13), the actual domains p_i passed to the class C being allocated are runtime domains, which K maps to static ODomains in PtD. We use H to lookup the OObject O_k for each value v_k passed to initialize the kth field of the object being allocated, and ensure that the OEdge is in PtE.

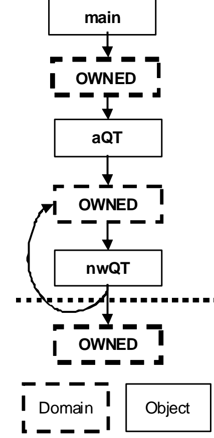
We define the approximation relation $\sim$ as follows:

$$
\begin{aligned}
&\forall \Sigma \vdash S, \quad (S, H, K) \sim (PtO, PtD, PtE) \\
&\text{iff } \forall \ell \in dom(S), \Sigma[\ell] = C\texttt{<}\overline{\ell'.d}\texttt{>} \\
&\text{implies} \\
&H[\ell] = O_C = \langle O_{id}, C\texttt{<}\overline{D}\texttt{>} \rangle \in PtO \\
&\text{and } \forall \ell'_j.d_j \in \overline{\ell'.d} \quad K[\ell'_j.d_j] = D_j = \langle D_{id_j}, d_j \rangle \in rng(PtD) \\
&\text{and } \forall d_i \in domains(C\texttt{<}\overline{\ell'.d}\texttt{>}) \\
&\qquad K[\ell.d_i] = D_i = \langle D_{id_i}, d_i \rangle \in rng(PtD) \\
&\qquad \text{and } \{(O_C, d_i) \mapsto D_i\} \in PtD \\
&\text{and } fields(\Sigma[\ell]) = \overline{T}\ \overline{f} \text{ and } \forall k, \forall \ell' \quad S[\ell, k] = \ell' \implies \\
&\qquad E_k = \langle H[\ell], f_k, H[\ell'] \rangle \in PtE
\end{aligned}
$$

Theorem: Object Graph Soundness.

$$
\begin{aligned}
&\forall G = \langle PtO, PtD, PtE \rangle \vdash P = (CT, e) \qquad CT, e \text{ well-typed} \\
&\forall e; \emptyset; \emptyset; \emptyset \rightsquigarrow^*_G e; S; H; K \\
&\forall \Sigma \ \vdash S \\
&PtO, PtD, PtE \vdash_{CT,H} \Sigma \\
&(S, H, K) \sim (PtO, PtD, PtE)
\end{aligned}
$$

The theorem states that, given any Runtime Object Graph (ROG) represented by a well-typed store S, and an OGraph produced from the same program P, there exists a map H that maps each location ℓ in the store to a unique OObject, and a map K that maps each runtime domain in the store to a unique ODomain, and this mapping is consistent with




```
Main main = new Main();
OObject(main, Main<null>)
analyze(main, [])
this ↦ main
class Main {
  domain D;
  ODomain(main.D, Main::D)
  OObject(main.D.aQT, QT<main.D>)
  QT<D> aQT = new QT<D>();
  analyze(main.D.aQT, [QT::M ↦ main.D])
  OEdge(main, main.D.aQT)
}
this ↦ main.D.aQT, [M ↦ main.D]
class QT<M> {
  domain D;
  ODomain(main.D.aQT.D, QT::D)
  OObject(main.D.aQT.D.nwQT, QT<main.D.aQT.D>)
  QT<M> nwQT = new QT<M>();
  analyze(main.D.aQT.D.nwQT, [QT::M ↦ main.D])
  OEdge(main.D.aQT, main.D.aQT.D.nwQT)
}
this ↦ main.D.aQT.D.nwQT, [M ↦ main.D]
class QT<M> {
  domain D;
  ODomain(main.D.aQT.D, QT::D)
  OObject(main.D.aQT.D.nwQT, QT<main.D.aQT.D>)
  QT<M> nwQT = new QT<M>();
  OEdge(main.D.aQT.D.nwQ, main.D.aQT.D.nwQT)
}
```

Figure 12. Example with recursive types.

IR-NEW

$$\frac{\begin{array}{c} \ell \notin dom(S) \qquad S' = S[\ell \mapsto C\text{<}\overline{p}\text{>}(\overline{v})] \\ \overline{p} = \overline{\ell'.d} \qquad D_i = K[\ell'_i.d_i] \\ O_C = \langle O_{id}, C\text{<}\overline{D}\text{>}\rangle \qquad O_C \in PtO \qquad H' = H[\ell \mapsto O_C] \\ \forall d_j \in domains(C\text{<}\overline{p}\text{>}) \\ D_j = PtD[(O_C, d_j)] \qquad K' = K[\ell.d_j \mapsto D_j] \\ \forall (T_k\ f_k) \in fields(C\text{<}\overline{p}\text{>}) \qquad O_k = H[v_k] \\ E_k = \langle O_C, f_k, O_k \rangle \qquad E_k \in PtE \end{array}}{\texttt{new}\ C\text{<}\overline{p}\text{>}()\overline{v}; S; H; K \rightsquigarrow_G \ell; S'; H'; K'}$$

Figure 13. Instrumented runtime semantics.

respect to the ownership relation. In addition, the OEdges in the OGraph soundly abstract all field points-to relations between any two objects in an ROG. More details and the proof are in [1, Chap. 3].

5. Architectural Abstraction

An extracted object graph provides architectural abstraction by ownership hierarchy and by types. But an object graph may not be isomorphic to an architect's intended architecture, and may require further abstraction.

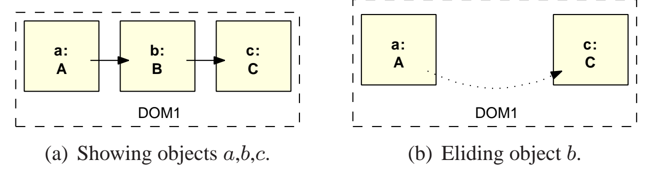
1. **Elide and summarize private domains.** Object graphs tend to expose the implementation of data structures [29, p. 252]. In SCHOLIA, when internal state is placed in private domains, the OOG abstraction step can leverage the semantic distinction between private and public domains.

For instance, the Aphyds designed architecture (Fig. 20) shows a circuit object, as well as node and net objects inside circuit. In the Aphyds object graph, the private domain `OWNED` on `Circuit` stores `Map`s of `Node` and `Net` objects (Fig. 15), and these objects are not architecturally significant. So the analysis, based on user input, can elide private domains and the objects they contain. To preserve soundness, however, the analysis may add *summary* edges to account for communication through elided objects. For example, if there is an edge from objects a to b and b to c, eliding b produces a *summary edge* between a and c (Fig. 14).

2. **Skip single domains.** In an OOG, each object is in a domain, so a systematic conversion would create each Component in a Group. Architects typically define tiers only at the top level, and those map to the top-level domains.

For example, requiring the Aphyds designed architecture to have a single DB tier inside circuit would be counterintuitive. Unless the developer requests otherwise, the conversion does not create a single tier inside a Component. Unlike eliding private domains, skipping single domains still creates the substructure for those unmapped domains. For example, after eliding the private domain `OWNED` inside `Circuit`, the conversion skips the single public domain DB and creates node and net and the connections between them, directly inside circuit (Fig. 23).

Even though domains play a central role in the annotations, they often disappear after they serve their purpose,



(a) Showing objects a,b,c. (b) Eliding object b.

Figure 14. Example of a *summary edge*.

which is to distinguish between internal and public state. Recall how in ownership domains, the owner of an object is a domain instead of another object, unlike other ownership type systems [10]. Indeed, both public and private domains produce hierarchy in an object graph. But we often elide private domains, end up with a single public domain in a given object, then skip that domain. Some type systems embody this idea and hard-code in each class, one private and one public *boundary* domain [31].

3. **Skip objects beyond a certain depth.** The analysis converts an OOG object hierarchy up to a user-selected depth, typically the depth of the hierarchical decomposition in the designed view. Reducing the size of the built architecture in this manner speeds up the comparison, but does not affect conformance, because lifted edges account for the elided substructures.

6. Architectural Description

SCHOLIA can represent the information that it reverse engineers from the code using different graphical (or non graphical) notations. Documenting an architecture in an architecture description language (ADL) enables performing various architectural-level analyses.

We use the Acme general purpose ADL [15], partly because of its available tool support. Acme represents architectural structure as a hierarchical graph with types and attributes on nodes and edges and has no execution semantics.

Most ADLs also support the following elements [25]. A Component is a unit of computation and state. A Port is a point of interaction on a Component. A Connector represents an interaction between Components. A System is a configuration of Components and Connectors. A Component can optionally be decomposed into a nested sub-architecture. A Property is a name and value pair associated with an element. A Group is a named set of elements, such as a tier.

To improve the precision of the structural comparison, the base architectural model has types and properties [6]. A Port that provides services has type ProvideT, and a Port that uses services has type UseT. The structural comparison uses the type information, when available, to avoid matching a ProvideT Port to a UseT Port, for example.

Components and Sub-Components. SCHOLIA assumes that an OOG has a single root. So the root object maps to a System. The top-level domains declared by the class of the root object map to the top-level tiers in the System. Each

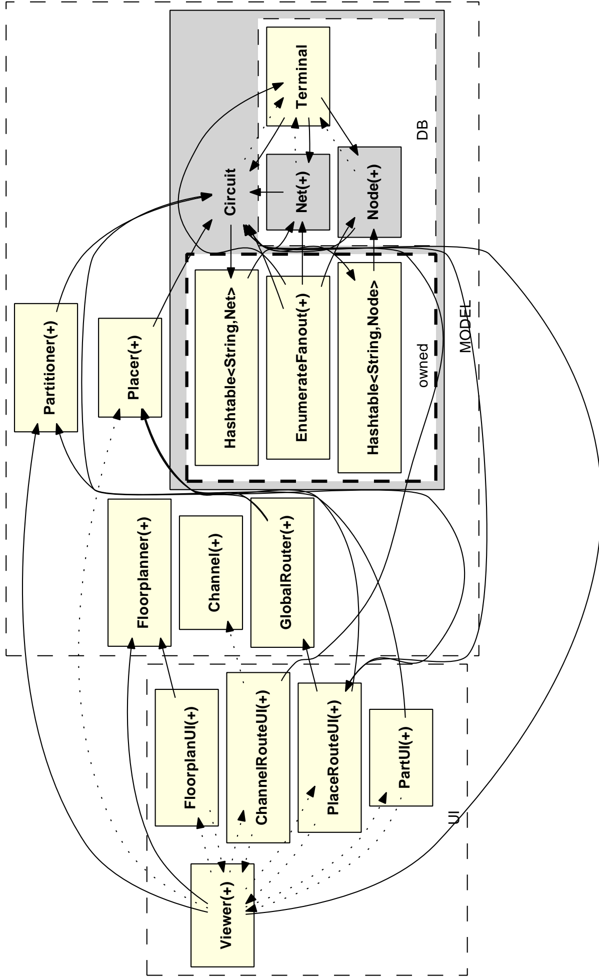
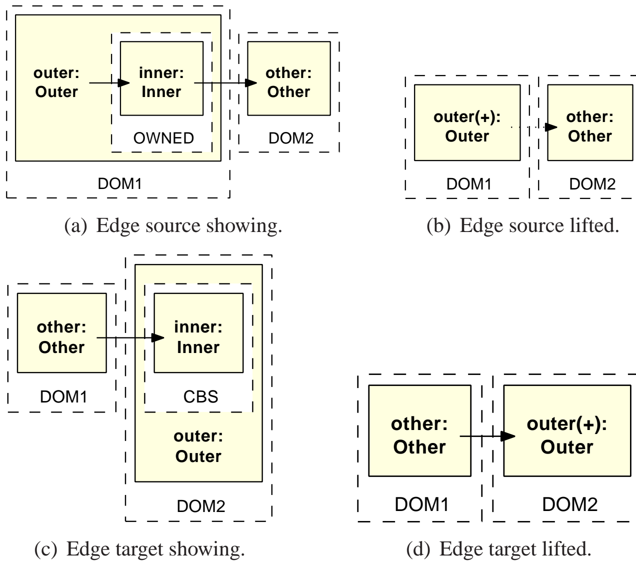


Figure 15. Aphyds OOG after defining public domains.



(a) Edge source showing.

(b) Edge source lifted.

(c) Edge target showing.

(d) Edge target lifted.

Figure 16. Examples of *lifted edges*.

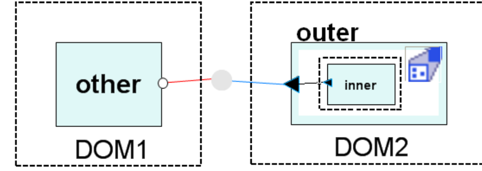


Figure 17. C&C view lifts edge to outer component.

object in the OOG maps to a Component. The OOG hierarchy creates architectural decomposition. If an OOG object declares domains and descendent objects, the corresponding Component has a sub-architecture.

Ports. References between objects create Ports as follows. If object `A` has a field reference of type `T` to object `B`, the corresponding Component A has a Port of type UseT and name B. The Component corresponding to `B` has a Port of type ProvideT and name `T`. And a Connector connects A to B. By default, the analysis does not represent the uninteresting self-edges in an OOG.

Edge Lifting. The representation of an OOG as a C&C view also lifts edges. Consider an OOG with an edge from `other` to `inner` inside `outer`'s public domain `CBS` (Fig. 16(c)). A C&C view lifts that edge to component outer, shows a connector from other to outer, and a connection from outer to inner (Fig. 17).

Domains and Tiers. An ownership domain d in the OOG maps to a Group g. If an object o in a domain d, the corresponding Component is in Group g. To be structurally comparable, both the built and the designed architectures follow similar topological constraints. For instance, in Acme, a Component can be included in more than one Group. But in ownership domains, each object is in exactly one domain and that domain never changes. So a predicate enforces that a Component or Connector is in exactly one Group. Moreover, if Connector c connects two Components that are in the same Group g, c must be also in g.

7. Architectural Conformance

SCHOLIA can just extract the up-to-date built runtime architecture from the code and document it an in ADL. If a documented target architecture exists, SCHOLIA can analyze its conformance with the code.

A designed architecture is often more abstract than the built architecture, but it must still represent all communication that could exist in the implementation. A conformance analysis can enforce the communication integrity principle and ensure that the designed architecture is a conservative abstraction of all the objects in the implemented system and the relations between those objects at runtime. A static analysis can of course suffer from false positives, and indicate potential object relations that can never exist at runtime. But here, the goal is to have no false negatives in the designed architecture, and show the worst case of possible communication between objects at runtime.

7.1 Analyzing and Displaying Conformance

In the terminology of Murphy et al. [28], the conformance analysis identifies:

- **Convergence:** a node or an edge that *is in both* the built and the designed architectures;
- **Divergence:** a node or an edge that is in the built architecture, but *not in the designed* architecture;
- **Absence:** a node or an edge that is in the designed architecture, but *not in the built* architecture.

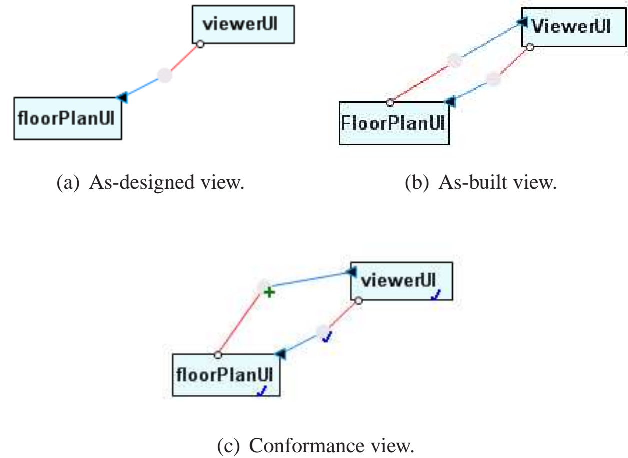
The analysis produces a *conformance view* as a copy of the designed architecture. The conformance view shows convergences and absences graphically, and represents divergences by showing additional connectors that are present in the implementation but are missing from the designed architecture. The analysis also sets various properties on the conformance view elements. Some of these properties decorate the graphical representation of an element. For instance, all elements have a finding property, set to convergent (shown as ✓), divergent (shown as +) or absent (shown as ✗).

As a positive side effect of the conformance analysis, SCHOLIA also establishes traceability between an intended architecture and the underlying source files, for the benefit of other code quality tools. The various steps thread through the traceability information as follows. The abstraction of an OOG into a C&C view copies the traceability of each OOG element into the traceability property of the corresponding C&C element, as a set of filename and line number pairs. Similarly, the conformance view derives its traceability information from the built C&C view. A tool can use this information in the conformance view to trace to the pertinent lines of code, and save a developer the effort of having to potentially review the entire code base to investigate a suspected architectural violation. Of course, the conformance analysis sets the traceability on only convergent and divergent elements, and not on absent ones.

The components an architect includes in the designed view may be more relevant than those she omits. And she often chooses names to convey her architectural intent. So, when analyzing conformance, SCHOLIA considers the designed view to be more authoritative than the built one, and works as follows:

1. **Match components, but use the names from the designed view.** Elements in the designed and the built views may not have exactly matching names. The structural comparison, however, can detect renames. Unlike view synchronization, the conformance analysis does not propagate the built names to the designed view.

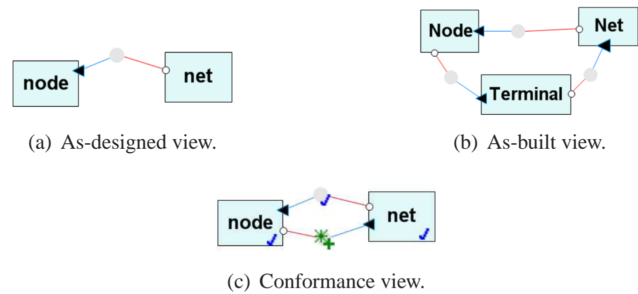
For Aphyds, the analysis correctly matches built components `ViewerUI` and `FloorPlanUI` to designed component viewerUI and floorplanUI, respectively, but does not rename them (Fig. 18).



(a) As-designed view. (b) As-built view.

(c) Conformance view.

Figure 18. Displaying a convergence and a divergence.



(a) As-designed view. (b) As-built view.

(c) Conformance view.

Figure 19. Showing a divergence as a *summary connector*.

2. **Highlight differing connections.** The analysis shows differing connections as divergences or absences. In Aphyds, the built view has only a connector between `FloorPlanUI` and `ViewerUI`, and the latter match the designed components floorplanUI and viewerUI. So the analysis shows a divergent connector from floorplanUI to viewerUI (Fig. 18). This requires the following stylized use of ports, which may also make ports easier to understand [8].

An Acme Port has no built-in directionality. Its type specifies whether it provides services (ProvideT) or uses services (UseT). In some cases, the designed view may have a connector between two components, but the connection in the built view may be in the reverse direction. The conformance analysis could make the Connector bi-directional, by assigning to the connection's endpoints both the ProvideT and UseT types. But this does not fit with showing divergences and absences. Instead, we adopt unidirectional ports, i.e., the type can be ProvideT or UseT, and never both. So the analysis shows a divergent connector, as well as ProvideT and UseT Ports, for the communication in the opposite direction.

3. **Summarize divergent components.** If there are components in the built architecture that are not in the designed architecture, the analysis works differently from view synchronization. Adding these components directly to the designed architecture would clutter it with implementation details. Instead, the analysis accounts for communication in

the built architecture that is not in the designed architecture, and may add *summary* connectors to abstract these divergent components and enforce communication integrity.

In the built view, Node connects to Terminal and Terminal to Net (Fig. 19(b)). The designed view has node and net, but has no component that matches Terminal (Fig. 19(a)). The analysis matches node to Node, and net to Net, respectively. It then shows a divergent connector from node to net, since the designed view does not already have one (Fig. 19(c)). If the designed view does have such a connector, the analysis marks it as convergent. Since a summary connector can be either divergent or convergent, the analysis sets a property isSummary on a connector separately from its finding. A decorator overlays the ✳ symbol on a connector when isSummary is set to true.

Viewed differently, the analysis represents using a summary connector any objects in the built view that do not have counterparts in the designed view. This allows a designed view to have a coarser granularity of components, and abstract multiple interacting objects with a connector. Indeed, the JavaDoc for Aphyds states that "Terminal is a *connection* between a Node and a Net".

To help a developer update an incomplete designed architecture, the analysis can optionally show in the conformance view the divergent components, but without showing any connections to these components. A developer can add some of the divergent components to the designed view and re-run the conformance analysis.

4. **Analyze matching substructures recursively.** Designed architectures are often hierarchical, but do not typically have deep hierarchies. An OOG provides architectural abstraction primarily through ownership hierarchy. When an OOG is abstracted into a C&C view — whether restricting the depth of the hierarchy or not, more components in the built C&C view will have substructures than their designed counterparts. To avoid generating many false positives, the analysis ignores the substructures that are in the built view but not in the designed one. Skipping unmatched substructures does not compromise soundness, because both an OOG (Figs. 16(b), 16(d)) and a built C&C view (Fig. 17) lift edges to represent any communication through their substructures.

For instance, viewerUI in the designed view does not define a substructure. So the analysis matches viewer to ViewerUI in the built view, and ignores the substructure of the latter. But the designed circuit has substructure and matches the built Circuit. In that case, the analysis recursively analyzes the substructures of circuit and Circuit. Had the OOG abstraction step not excluded private domains, the conformance analysis would have processed the corresponding OWNED tier in the built C&C view, and generated several undesired divergences, since both domains OWNED and DB are in Circuit's substructure, and its designed counterpart also has substructure.

7.2 Measuring Conformance

SCHOLIA counts convergent edges (CE), divergent edges (DE), absent edges (AE), and summary edges (SE). In addition, SCHOLIA counts convergent nodes (CN), divergent nodes (DN), and absent nodes (AN). In SCHOLIA, a high AN or DN often indicate that the designed view is missing components compared to the built view, or uses a different system decomposition (Table 1).

SCHOLIA combines edge divergences and edge absences into one number. In terms of face validity, this metric is similar to a *graph edit distance*, which models inconsistencies by transforming one graph into another [12]. Typical edit operations include the deletion, insertion and relabeling of nodes and edges. Each edit operation is assigned an application-dependent cost. SCHOLIA assigns renames a zero cost and counts insertions (divergences) and deletions (absences).

The Core Conformance Metric (CCM) counts divergent edges (DE) and absent edges (AE) that would make the designed architecture account for all communication in the implementation. To get a percentage, we divide by the total number of edges and subtract from 100%. Of course, fewer absences and divergences are better and mean the system is closer to the target architecture. So, a higher CCM value indicates a higher structural conformance.

$$\mathbf{CCM} = 1 - \frac{\mathsf{AE} + \mathsf{DE}}{\mathsf{CE} + \mathsf{AE} + \mathsf{DE}}$$

SCHOLIA qualifies the conformance metrics by measuring the percentage of the program that lacks annotations. For simplicity, SCHOLIA uses a derived measure, WARN, namely the number of annotation warnings that the annotation typechecker generates. Except for some defaults, every field, variable declaration, or method return, that is a reference to an object and has a missing or incorrect annotation, generates a warning (we mostly avoid multiple warnings due to one missing annotation). To get a percentage, the metric WARN% normalizes WARN by the number of declared object references in the program. Thus, WARN% is an indicator of how many annotations are missing to make an OOG soundly represent the built architecture. A lower WARN% is better. For a program without annotations, WARN% will be high. As valid annotations are added, or warnings are addressed, WARN% decreases.

For Aphyds, WARN% is 5%. The remaining warnings are due to expressiveness challenges in the type system, which we discuss elsewhere [2]. We believe however these warnings do not contribute to missed architectural violations.

8. Evaluation

Our evaluation demonstrates the *feasibility* of SCHOLIA and that hierarchical object graphs provide architectural abstraction, something that had been missing in previous static analyses of the runtime structure. In future work, we plan to eval-

uate SCHOLIA's *usefulness*, i.e., if it can provide actual assistance to a developer in fulfilling a code modification task based on an object diagram, as well as the *usability* or ease of learning and applying the approach.

Research question. The evaluation aimed to answer the research question: *Can* SCHOLIA *identify interesting structural differences between built and designed architectures in real systems?* A finding is interesting if it identifies undocumented information, contradicts available documentation, or highlights a potential design or implementation defect. We refine the research question into the following hypotheses:

- *A developer can control the annotations to extract a built architecture that expresses his architectural intent and conveys architectural abstraction.* The measurable criteria are to minimize annotation warnings, reduce the number of top-level objects compared to a flat object graph, and not display low-level objects.
- *The conformance analysis can match the built and the designed architectures, display a readable conformance view, enable tracing a finding to the code, and compute sensible conformance metrics.* The measurable criteria are to minimize false positives and to be able to trace to the right code locations.

Methodology. A developer documents the designed architecture in an ADL. She then adds annotations to the code, invokes a typechecker and addresses annotation warnings.

Just as there are multiple architectural views of a system, there is no single right way to annotate a program. Good annotations minimize the number of top-level objects, by pushing low-level objects underneath more architecturally-relevant ones. For a meaningful comparison, the designed and the built architectures must have similar tiers, similar hierarchical decomposition, and similar components and tiers at each hierarchy level.

Using a tool, she extracts a hierarchical object graph, and refines the annotations until the number of top-level objects is roughly comparable to that in the designed architecture. She then invokes a tool to abstract the extracted object graph into a built architecture. She then uses another tool to compare the built and the designed architectures. She typically only confirms the results of the comparison. But if the comparison mismatches some elements, she can manually force or prevent matches between those elements, and rerun the comparison. Finally, she examines the results of the conformance analysis, studies unexpected findings and traces suspicious ones to the code.

The developer can iteratively: (a) refine the annotations; (b) manually guide the comparison if it fails to perform the proper match; (c) correct the code, if she decides that the designed architecture is correct, and the implementation violates the architecture; or (d) update the designed architecture if she considers that the implementation highlights an error or omission in the target architecture.

Tools. To support the methodology, SCHOLIA uses several Eclipse plugins to relate C&C views, OOGs and source files:

- AcmeStudio is an Acme modeling environment [15], to document the designed architecture and display the conformance view. AcmeStudio is an Eclipse perspective, so a developer can trace seamlessly from a conformance view to the Java code in Eclipse;
- ArchDomJ typechecks the annotations added to the code as Java 1.5 annotations and displays warnings in the Eclipse problem window. A developer can go from a warning to the offending line of code;
- ArchRecJ extracts an OOG from annotated code;
- ArchCog abstracts an OOG into a C&C view (Section 5). A developer can elide private domains or restrict the projection depth;
- ArchConf analyzes conformance between two C&C views, generates a *conformance view* and computes the metrics (Section 7). ArchConf allows a developer to confirm the results of the structural comparison, or to manually force or prevent matches and rerun the comparison;
- CodeTraceJ loads the traceability of an element in the conformance view, opens the corresponding source files and highlights the appropriate lines;
- ArchMod modifies the original designed architecture, by taking a divergent element from the conformance view and adding it to the designed view, or deleting an absent element from the designed view.

Aphyds case study. We now describe analyzing the conformance of the Aphyds system using the above methodology and tools. The experimenter (one of us, hereafter "we") developed several of the tools, but none of the subject systems. The process was iterative as a whole, and involved both macro- and micro-iterations. A macro-iteration consists of documenting the designed architecture, adding the annotations, extracting an OOG, abstracting it into a built C&C view, and analyzing its conformance. A micro-iteration can consist of iterating the annotations and the OOG extraction before converting the OOG into a C&C view, until the OOG has a reasonable abstraction level, e.g., by abstracting away low-level objects such as `Vectors` from the top-level domains. Retrospectively, we present our evaluation as two macro-iterations, and show the evolution of the conformance metrics across the two macro-iterations (Table 1).

Designed architecture. We formalized the Aphyds designed architecture based on the informal diagram (Fig. 20), but iterated it a few times while formalizing it. When connecting two components in a group, we initially forgot to put the connector into that group, which resulted in the conformance analysis badly matching those connectors.

In an early iteration, we set the analysis to add the divergent components to the conformance view, and noticed a partitionUI component. For consistency, since floorPlanUI and placeRouteUI interact with floorplanner and placeRouter, respectively, we added to the designed architecture

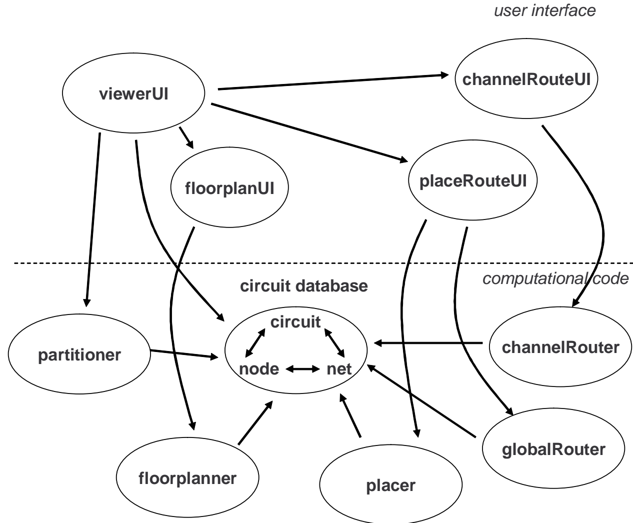


Figure 20. Aphyds designed architecture, redrawn from the original developer's diagram. User interface components such as viewerUI are in the upper half. A circuit and computational components are the lower half. Here, edges represent points-to relations.

a partitionUI that interacts with partitioner, even though the informal drawing omitted partitionUI.

Iteration 1. We initially organized the Aphyds objects into two top-level domains, `UI` and `MODEL`. `UI` holds a `ViewerUI` object and several subsidiary user interface objects. `MODEL` holds a `Circuit` object and computational objects that act on it, such as `Floorplanner`. We also defined several private domains to hold objects encapsulated by their parent, such as `Map` objects inside a `Circuit` object, as the (+) sign indicates in Fig. 21. These annotations produce a hierarchical OOG that has many objects in the top-level domains.

Conformance metrics. The conformance analysis does not produce good conformance metrics (Table 1). For example, `Node` and `Net` are peers of `Circuit` instead of being in its substructure (Fig. 22). So the conformance analysis marks as absent the node and net components inside circuit, hence the 2 node absences.

The built view has many more components in the top-level tiers than the designed view, which explains the high node divergence. Moreover, the conformance analysis generates many summary connectors to account for possible transitive communication, which leads to a high number of edge divergences and an unreadable conformance view.

For example, `Displayer` communicates with `Terminal`, and `Terminal` with `Placer`. In reality, `Terminal` is part of `Circuit`, and `Circuit` already communicates with `Placer`. Ideally, the analysis should just mark as convergences the connection between `Displayer` and `Circuit`, and the one between `Circuit` and `Placer`. Since the analysis lacks information about logical containment, it shows instead a divergent summary connector from `Displayer` to

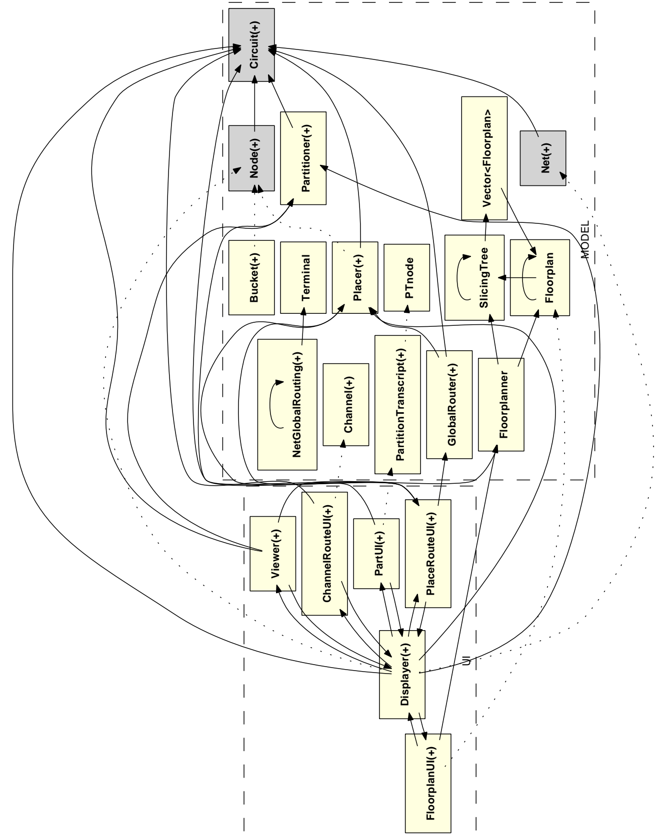


Figure 21. Aphyds OOG using private domains and many peer objects, e.g., `Node`, `Net`, `Terminal` and `Circuit`.

```
1  class Circuit<OWNER> { // Implicit parameter
2    domain OWNED; // Private domain
3    OWNER Node node; // Make peer to self
4    OWNER Net net;
5    OWNER Terminal terminal;
6    // The outer OWNED annotation is for the Map object
7    // The inner OWNER annotation is for the map elements
8    // String objects have manifest ownership
9    OWNED Map<String, OWNER Node> nodes;
10 }
11 // Everything else is exactly the same as Fig. 5
```

Figure 22. Initial annotations.

`Placer`, and many others. This turns the conformance view into an unreadable fully-connected graph. The low CCM and the many summary edges (SE) — 97 in total, may not mean that the designed view is only 21% accurate, but that the built architecture is not yet meaningfully comparable to the designed one.

In SCHOLIA, a developer controls the architectural abstraction using annotations. So in the second iteration, we refined the annotations to get a better match, without changing the code. The reader can visually compare the annotations

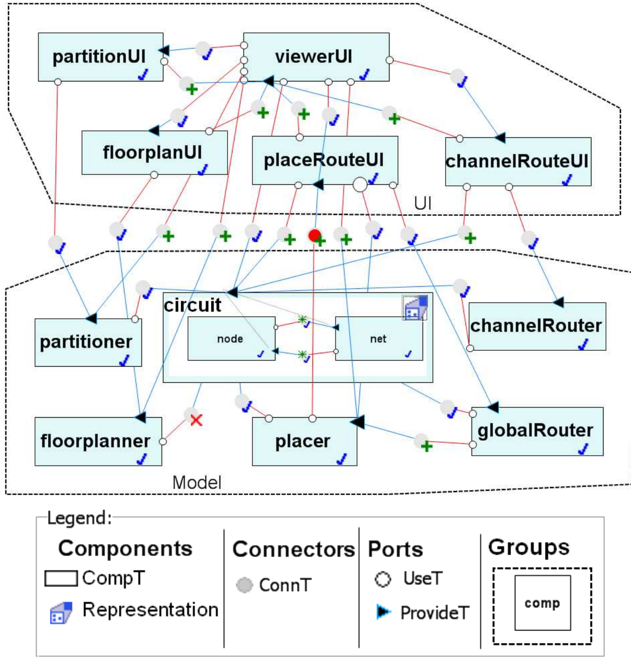


Figure 23. Aphyds conformance results.

in Fig. 22 which we used in Iteration 1, to those in Fig. 5, which we adopted in Iteration 2.

Iteration 2. Using the designed architecture as a guide (Fig. 20), we defined several public domains to logically contain objects that should not be in the top-level domains. For example, `Viewer` has a `DISPLAY` public domain to hold a `Displayer` object. `Displayer` is not in the developer's diagram (Fig. 20), but is not encapsulated either. `Displayer` is only logically contained inside `ViewerUI`, and many other `UI` objects such as `FloorPlanUI` reference it directly.

Other public domains abstract low-level objects into more architecturally relevant ones. For example, `Circuit` holds objects such as `Node` and `Net` inside its `DB` public domain, to reflect the designed architecture (Fig. 20).

In most cases, defining public domains required mostly local and incremental changes to the annotations. With the refined annotations, many objects that were in the `MODEL` top-level domain, such as `Node`, `Net` and `Terminal`, moved into public domains of other objects, such as `Circuit` (Fig. 15). As a result, both the extracted OOG and the abstracted built view now have a system decomposition that is closer to the desired architecture (Fig. 20).

Conformance metrics. Iteration 2 matched the components better, with 0 node absences and 1 node divergence, which corresponds to `Terminal`. The analysis now marks as convergent, both node and net inside circuit, as well as the connectors between them (Fig. 23). In the built system, node and net do not communicate directly, but only do so through `Terminal`. So the two convergent connectors inside circuit have the summary decoration ✳. As an aside, the edges from `Node` to `Terminal` and from `Net` to `Terminal`

Table 1. Aphyds conformance metrics.

Iteration	CN	DN	AN	CE	DE	AE	SE	CCM
1	11	11	2	23	89	0	97	21%
2	13	1	0	16	11	1	2	57%

are in fact lifted edges. This example justifies the different kinds of edge summarization, such as edge lifting in a C&C view, then adding summary connectors in the C&C view.

Overall findings. As one would expect from an informal diagram, the designed architecture (Fig. 20) is only about 60% accurate, based on the CCM metric. Indeed, SCHOLIA identified a divergent component partitionUI, several divergences between viewerUI and other UI components, between UI and MODEL components, and between MODEL components. Many connections which the developer thought to be uni-directional were bi-directional in reality.

One divergence that crosses tiers, from placer in `MODEL` to placeRouteUI in `UI`, was a red flag (this is the connector we manually set to be darker in color in Fig. 23). A multi-threaded application must respect certain framework-specific conventions to call back from a worker thread executing a long-running operation into the user interface thread. We used CodeTraceJ to trace this divergence to a `PlaceRouteUI` field inside class `Placer`, and checked that the Aphyds code handled this callback correctly.

Tool performance. The tools are sufficiently interactive to allow iteration. On an Intel ® Core™ 2 Quad Processor (2.4 GHz) with 4GB of RAM running Windows XP, the OOG extraction takes around 10 seconds, and the structural comparison takes between 57 seconds (Iteration 1) and 33 seconds (Iteration 2).

9. Discussion

Internal threats to validity may indicate that factors other than the technique determined the results. *External threats* limit the extent to which the results can be generalized.

Internal validity. One threat to internal validity is that, even though we did not author Aphyds, we previously studied it in various ways [6, 8]. We believe the results of this case study are due to using SCHOLIA and not to any previous knowledge of the code. The code base is non-trivial enough for anyone to memorize. Moreover, we previously represented the desired architecture differently [6, Fig. 19]: we did not consider tiers, had one `model` component with planner, partitioner and others as sub-components, and ignored circuit's substructure. Although the experimenter also designed several of the tools, a typechecker kept him honest. He could not insert an arbitrary annotation without getting a warning, or otherwise manipulate the extracted architectures.

Another threat is that an electrical engineering professor, not a professional architect, drew the Aphyds intended architecture. However, we only mined the diagram for the architecturally significant objects and tiers it shows, and for

the hierarchical system decomposition it uses for circuit, all general concepts in modeling architectures [11].

Another confound is whether the built and the designed architectures represent the same information. For instance, when we redrew the original developer's diagram (Fig. 20), we reversed the direction of some arrows [8, p.192] and excluded data flow edges. For a meaningful conformance analysis, the designed and the built architectures must have the same kind of connectors, here, points-to relations.

Can SCHOLIA *identify at least as many violations as the state-of-the-art in the static enforcement of runtime architectures?* The state-of-the-art would be library-based [24] or language-based [8, 32] solutions. For instance, the C2 ADL mandates a specific architectural framework [24], but requires developers to follow strict guidelines to avoid introducing architectural violations. There are no tools to check that an implementation obeys those rules (N. Medvidovic, personal communication, 2008). Language-based solutions, first exemplified by ArchJava, radically extend the language to incorporate architectural components and ports, and enforce communication integrity using a type system [8, 32].

Aldrich et al. previously studied Aphyds and identified similar architectural violations, but only after they re-engineered it to ArchJava [8]. ArchJava specifies in code architectural hierarchy and instances. In ArchJava, an object is architecturally significant if its declared type is a `component class`. However, in ArchJava, a method can neither take as an argument, nor return a reference to an instance of a `component class`. Because real object-oriented code passes around object references liberally, using ArchJava in an existing Java code base is harder than simply converting each Java `class` into an ArchJava `component class` [5]. Adopting ArchJava often requires a non-trivial re-engineering that changes how objects are passed around. When using ArchJava, one may define additional `component classes` to capture the intended system decomposition. For Aphyds, Aldrich et al. specified 20 ArchJava `component classes` and over 80 `ports`, re-engineered the program to obey ArchJava's restrictions, and inadvertently injected defects [8].

SCHOLIA achieves hierarchy using annotations and without additional classes. In SCHOLIA, all objects are instances of regular Java classes, and there are no restrictions on passing object references. The more architectural objects are higher in the ownership hierarchy. In particular, logical containment can impose an arbitrary hierarchy on an object graph, and allows SCHOLIA to support arbitrary object-oriented code better. Of course, specifying strict encapsulation to avoid the representation exposure may require a change to the code, e.g., to return a copy of an internal list instead of an alias [7]. During our Aphyds evaluation, we only added annotations.

Could any other static approach find the violations that SCHOLIA *found?* It is a genuine threat to validity to compare a designed runtime architecture to a built code architecture, or vice versa. All previous *static* conformance approaches, e.g., [28], address the *code architecture*. The closest to a *statically* extracted *runtime architecture* for an object-oriented system would be an object graph extracted by a static analysis, whether it uses annotations [20] or not [17, 29]. All previous graphs — with the exception of our own previous work [4] — are flat, and would not convey enough architectural abstraction to enable conformance analysis. Of course, we could compare SCHOLIA's results to those obtained by a *dynamic* analysis [34, 33]. But a dynamic analysis cannot claim to represent all possible executions.

Could a conformance analysis of the code architecture detect all the violations in a runtime architecture? For example, *could Reflexion Models (RM) [28] find all the violations that* SCHOLIA *found?* In fact, we modeled SCHOLIA closely after RM, which is a standard bearer in analyzing the conformance of code architectures. In RM, a third-party tool extracts a *source model* from the implementation. A developer posits an as-designed *high-level model* and a *map* between the source and high-level models. RM pushes each interaction described in the source model through the map to infer edges between high-level model entities. RM then compares the inferred edges with the edges stated in the high-level model.

There are similarities between SCHOLIA and RM. For example, WARN is similar to how RM tracks unmapped entries in the source model. A major difference is that RM is designed for the code architecture. There are also several minor differences. For example, RM has no divergent or absent nodes. In RM, if the map generates a node that is not the designed view, RM automatically adds that node to the designed view. In other words, RM has no divergent or absent nodes, nor does it compute summary edges. To our knowledge, other static conformance checking techniques of the code architecture are not more expressive than RM.

In Aphyds, many important classes are instantiated once, so for those classes, the object graph is somewhat similar to a class diagram with associations. Of course, there are still non-trivial differences related to the different instantiations of the various container classes such as `Vector`. Out of curiosity, we ran jRM [18] on Aphyds. jRM supports neither tiers nor hierarchical target architectures, so we used a simplified high-level model without tiers and ignored `Circuit`'s substructure. Indeed, RM found the divergence from placer to placeRouteUI, because it corresponds to a direct field reference declared in class `Placer`. However, RM showed *absences* between viewerUI and floorPlanUI instead of the correct divergences and convergences (See RM's output for Aphyds in [1, Chap. 7]).

In the OOG, a `ViewerUI` object does not directly point to a `FloorPlanUI` object. Instead, a `ViewerUI` points to a `Displayer`, and `Displayer` references a `FloorPlanUI`. Moreover, `Displayer` is in a public domain of `ViewerUI`.

When `ViewerUI`'s substructure is elided, the OOG *lifts* that relation to `ViewerUI`, and shows a *lifted edge* from `ViewerUI` to `FloorPlanUI`, shown as a dotted edge in the OOG (Figs. 21, 15).

Similarly, RM would not correctly handle circuit's substructure, such as the communication between node and net. Unlike RM, SCHOLIA distinguishes the `Vector` of `Terminals` inside `Net` from the one inside `Node`, and this distinction produces the communication between `Node`, `Net` and `Terminal`. Then, SCHOLIA represents the communication between node and net through edge lifting and summary connectors. Thus, in general, a tool for the code architecture cannot handle the runtime architecture.

Does SCHOLIA *generate many false positives?* False positives are possible in general, but SCHOLIA attempts to reduce them. For example, the edges in an OOG are more precise than super-imposing associations from a class diagram. Also, SCHOLIA analyzes only matching substructures, and not the entire object hierarchy. There are several sources of false positives in SCHOLIA. The OOG extraction uses a whole-program and not a reachability analysis that excludes infeasible paths. Also, the conformance analysis may add summary edges that are false positives, as in the first iteration which had 97 summary edges. But if the built and the designed architectures have a similar hierarchical decomposition and a similar number of components at each hierarchy level, the analysis adds fewer summary edges. Indeed, the second iteration had only 2 summary edges, and neither one was a false positive. In our Aphyds evaluation, we used CodeTraceJ to trace each finding to the code, and confirmed that it does not correspond to an obvious false positive. Aphyds was written by a professor for one of his classes. So this may explain the absence of infeasible paths.

External validity. *Can* SCHOLIA *find architectural violations in other systems?* Yes. We have applied SCHOLIA to two other systems. Due to space limits, we highlighted here the Aphyds evaluation. The others are available in the first author's dissertation [1, Chap. 7]. JHotDraw (15 KLOC) is designed by experts in object-oriented analysis and design. HillClimber (15 KLOC) is designed by undergraduates, and was previously re-engineered to ArchJava to specify its architecture [5]. We also added annotations to, and extracted OOGs from LbGrid, a 30-KLOC module that is part of a 250-KLOC commercial system [3]. The architects did not provide us, however, with a designed runtime architecture, so we could not analyze it.

In all the architectures we analyzed, SCHOLIA found many omitted components or connections. For example, the JHotDraw architecture omitted components that were added later to support undoing commands.

Can SCHOLIA *analyze architectures that specify fine-grained object structures or multiplicities?* An OOG and its abstracted C&C view provide architectural abstraction by merging equivalent instances in a domain or tier. So SCHOLIA cannot express very fine-grained object structures. Similarly, as with most static object diagrams, SCHOLIA does not provide any precision regarding multiplicities.

Would an outside developer understand the SCHOLIA *technique?* Until there are better tools for adding annotations, our approach does not have the characteristic of Reflexion Models that third-party users can easily run on large code bases [28]. As a result, a study with an outside developer would be difficult given the nature of the approach. We did, however, conduct a field study and confirmed that, indeed, an outside professional programmer understood abstraction by ownership hierarchy and by types [3].

Admittedly, the need to iteratively improve the annotations, fine-tune how an OOG is abstracted into a C&C view, and follow all the steps in the tool chain may be a challenge to the average developer. However, this situation is not unique to SCHOLIA. For example, previous work on code architectures using semi-automated clustering algorithms, required that developers spend significant effort fine-tuning the clustering parameters to derive a good match [9]. In SCHOLIA, a developer uses annotations to control the abstraction and does not rely on a tool's hard-coded heuristics.

Is SCHOLIA *more lightweight than other static conformance approaches?* For example, *is adding ownership annotations to an existing system less invasive than re-engineering it to ArchJava to expose its architecture?* Our preliminary evidence showed that to be the case [5]. The annotations, unlike ArchJava, do not change the system's runtime semantics, and support common object-oriented idioms, such as passing references to objects. For example, an ArchJava `component class` cannot have `public` fields. When using ownership annotations, such legal Java fields can be placed in public domains. Aldrich et al. added ownership types to the model part of Aphyds (3.5 KLOC) in 4 hours, a quarter of the time they spent re-engineering that same part to ArchJava [8].

To more reliably estimate the annotation effort, we conducted a week long on-site field study. The first author spent 35 hours adding annotations and extracting OOGs from the 30-KLOC LbGrid module (WARN is still high). Based on our previous experience with ArchJava [5], we could not have re-engineered LbGrid to ArchJava in the same few days that it took us to add the annotations, even after accounting for possible tool and language familiarity. Thus, adding annotations to an existing system seems more lightweight than re-engineering it to use an extended language like ArchJava.

Would SCHOLIA *work with an ownership type system other than ownership domains?* In principle, SCHOLIA could use a type system that assumes a single *context* per object [10]. There is, however, a crucial expressiveness advantage in ownership domains that can reduce the number of objects in the top-level domains. In an *owner-as-dominator* type system, any access to a child object must go through its owning object [10]. In contrast, ownership domains sup-

port pushing almost any object underneath any other object in the ownership hierarchy. A child object may or may not be encapsulated by its parent object: a child object can still be referenced from outside its owner if it is part of a public domain of its parent, or if a domain parameter is linked to a private domain [7]. SCHOLIA can readily use an ownership type system such as Simple Loose Ownership Domains [31], which enforces a *boundary-as-dominator* property.

For arbitrary object-oriented implementation code, it is easier to use logical containment with public domains, rather than the strict encapsulation of private domains — and both can reduce the number of objects in the top-level domains.

Why structural comparison? SCHOLIA compares the designed and the built architectures using a structural comparison that works with hierarchical views, does not assume unique identifiers, detects renames and allows forcing or preventing matches between selected view elements. These assumptions closely match the problem of analyzing conformance after the fact. SCHOLIA does not assume that the architectural components have unique identifiers, which would simplify the graph comparison considerably [12]. Using structural comparison enables SCHOLIA to detect renames between the built and the designed architectures, which can partly occur due to the OOG extraction.

The OOG extraction nondeterministically selects a label for a given object o based on the name or the type of one of the references in the program that points to o. Thus, detecting renames ensures a developer can still rename fields or local variables or types without impacting conformance. Avoiding the rename problem would require additional annotations to specify in code the displayed labels.

Assumptions. SCHOLIA makes the following assumptions:

- **Sources available:** The program's whole source code and portions of external libraries that are in use have annotations that typecheck;
- **Single entry point:** The program operates by creating a main object. The class of that object declares domains, but has no domain parameters;
- **Summarized external entities:** Reflection, dynamic code loading or native calls may introduce unknown objects and edges into the system. Such external entities can be summarized using "virtual field" annotations [2].

Limitations. SCHOLIA has the following limitations:

Annotations. The manual annotation effort is a potential obstacle for practical adoption, but ownership annotations are amenable to automated ownership inference, which could alleviate this problem, at least partially [23]. With precise and scalable ownership inference, SCHOLIA can scale to large systems.

Architectural extraction. SCHOLIA applies to applications that run in a single virtual machine, so it handles neither heterogeneous nor distributed systems, nor does it address dynamic architectural reconfiguration.

Structural comparison. If the views are very different, an automated structural comparison may fail to match the built and the designed views. In that case, the comparison will not be useful since all components will be absences. One can then manually match some view elements at the cost of additional effort. Finally, the algorithm is quadratic in the view sizes. So, while it scales to up to a few thousand nodes [6], very large architectures may be intractable.

10. Related Work

View synchronization. Our conformance analysis specializes our view synchronization work [6]. The key changes include: (a) processing the view differences more selectively (Section 7), such as skipping unmatched hierarchical decompositions, instead of making the two views identical; (b) computing summary connectors; and (c) including tiers in the hierarchical data used by the structural comparison, i.e., a Component or a Connector is a child of its owning Group. We observed empirically that this extra level of hierarchy improves the precision of the structural comparison, and enables it to distinguish better the connectors within a given tier (which would belong to the same Group) from the ones that cross tiers (which would not be inside a Group).

Code architecture. Several approaches analyze the conformance of code architectures, e.g., [28]. Generally, an approach designed for the code architectures, e.g., [28], cannot handle correctly the runtime architecture. However, several techniques we used, namely hierarchy, lifted edges and summary edges, have previously been applied to code architectures. We showed how the above techniques translate naturally to runtime architectures.

Hierarchy. Reflexion Models (RM) uses non-hierarchical high-level models and maps. Koschke et al. extended RM with hierarchical models [19]. In SCHOLIA, all the representations are hierarchical.

Lifted edges. Approaches that handle code architectures also lift edges [19, 37], for example, from a function call to a module. We use edge lifting in several places: an OOG lifts object relations from child objects to their parents; and a C&C view also lifts edges from inner components (Fig. 17).

Summary edges. Ommering et al. create a second module view that displays the transitive closure of a relation in one module view [37]. Our summary edges (Fig. 14) or summary connectors (Fig. 19) show transitive communication.

Dynamic analyses. Several approaches uses dynamic analysis to extract the built architecture [33, 13] or monitor conformance [22, 34]. For example, DISCOTECT [33] recovers from a running system a built C&C view that has architectural types. In place of annotations, DISCOTECT requires rules that map entries in a runtime trace to architectural events, e.g., a method invocation leads to the creation of a port. In DISCOTECT, it may be possible to reuse a mapping across several similar systems, which is not the case with our annotations. Because DISCOTECT is a dynamic analysis, the

results reflect only the particular inputs and exercised use cases. Also, DiscoTect generates non-hierarchical C&C views that show one component for each instance created at runtime. Finally, DiscoTect only extracts built views and does not analyze conformance.

Several dynamic analyses infer hierarchical object graphs without using annotations, e.g. [16], but their results describe only the structure for those program runs. They also adopt restrictive notions of ownership which cannot express many design idioms. The expressiveness in ownership domain avoids a built architecture with many components in the top-level tiers. Our evaluation showed how crucial that can be for a meaningful conformance analysis.

Code generation. Some approaches assume that developers always refine an architectural model into code to ensure conformance by design. Scholia is designed to analyze the conformance of an arbitrary system after the fact, requiring only annotations.

Static analysis. Lam and Rinard proposed a type system and a static analysis (LR) that uses non-ownership annotations to extract non-hierarchical object graphs [20] (LR does not analyze conformance). LR supports a fixed set of statically declared global tokens, and the result of the analysis is a graph showing which objects appear in which tokens. Using token parameters, the same code element can be mapped to different design elements depending on context. Unlike ownership domains, LR has a statically fixed number of tokens, all at the top level, so LR cannot show hierarchy. For Aphyds, LR would produce an object graph with even more top-level objects than Fig. 21, which would make it even less suitable for conformance analysis.

Our previous work. We previously presented an earlier definition of the extraction static analysis, using an alternate formalization based on rewriting rules [4]. This paper's version is different in several respects. Here, we use abstract interpretation, which makes the analysis more comparable to previous points-to analyses. The soundness proof now includes edges. This more principled formalization side-steps determining a depth at which to cutoff the recursion and the potential unsoundness of selecting an incorrect depth. The earlier system proved partial soundness on an intermediate cyclic representation, which is then projected or unfolded into a graph that the user sees.

Points-to analysis. All previous points-to analysis produce non-hierarchical graphs [36, 27]. Our static analysis is similar to a flow-insensitive Andersen-style points-to analysis. The state-of-the-art is considered an *object-sensitive* analysis [27]. Our analysis is object-insensitive but can be considered *domain-sensitive*, since it distinguishes between objects in different domains. Since domains are coarser-grained than objects, our analysis is more scalable than an object-sensitive one. However, our analysis suffers from some of the imprecisions that object-sensitivity addresses such as field assignment through a superclass [27].

Although points-to analysis is often used for compiler optimization, its value for program understanding has been recognized [36]. In the same vein as Scholia, Milanova [26] uses the results of a points-to analysis to construct an Object Relation Diagram, which is a class diagram where the type of the pointed-to object is potentially more precise than the declared type. To our knowledge, Scholia is the first approach to abstract the output of a static points-to analysis into a hierarchical runtime architecture represented as a standard Component-and-Connector (C&C) view, then using that to analyze conformance of runtime architectures.

Shape analysis. Shape analysis, e.g., [30] produces very precise shape graphs consisting of nodes to represent a set of objects, and edges to represent points-to relations. However, a shape graph is non-hierarchical: all the nodes are at the same level, and objects are not collapsed underneath other objects. This works well in an intra-procedural case to show that a method preserves the list-ness of a data structure it takes as a parameter. Moreover, a heavyweight shape analysis may also achieve more precision than Scholia in many cases. But a flat object graph will not scale to an entire system. Although Scholia sacrifices some precision to gain scalability of the analysis, it conveys architectural abstraction primarily through hierarchy.

11. Conclusion

Scholia is the first approach to extract statically a hierarchical runtime architecture from a program in a widely used object-oriented language, using annotations. If an intended architecture exists, Scholia can also analyze, at compile-time, communication integrity between the code and the target architecture. In practice, Scholia found interesting structural differences between existing systems and their target architecture. Our evaluation confirms what others have reported [28, 8], that informal diagrams often omit important communication. Thus, analyzing conformance after the fact is practically relevant during software evolution.

Finally, Scholia can establish traceability between an implementation and an intended runtime architecture. To our knowledge, Scholia is the first approach that allows a developer to trace from an element such as a component or a port in a runtime architecture, extracted entirely statically, to the corresponding lines of code in a general purpose object-oriented language like Java. This facility was available only when tracing from UML class diagrams to Java code.

Until now, developers evolving an object-oriented system had to contend with high-level views of the code architecture or partial views of the runtime architecture obtained using dynamic analysis. Scholia now completes the picture.

Acknowledgments. This work was supported in part by Aldrich's NSF CAREER award CCF-0546550, DARPA contract HR00110710019, and Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems."

The authors thank the other Ph.D. thesis supervisors, Nenad Medvidovic, Brad A. Myers, and William Scherlis, as well as David Garlan, Bradley Schmerl and Mary Shaw.

References

[1] M. Abi-Antoun. *Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure*. PhD thesis, Carnegie Mellon University. Available as Technical Report CMU-ISR-09-119.

[2] M. Abi-Antoun and J. Aldrich. Ownership Domains in the Real World. In *IWACO*, pages 93–104, 2007.

[3] M. Abi-Antoun and J. Aldrich. A Field Study in Static Extraction of Runtime Architectures. In *PASTE*, 2008.

[4] M. Abi-Antoun and J. Aldrich. Static Extraction of Sound Hierarchical Runtime Object Graphs. In *Types in Lang. Design and Impl. (TLDI)*, pages 51–64, 2009.

[5] M. Abi-Antoun, J. Aldrich, and W. Coelho. A Case Study in Re-engineering to Enforce Architectural Control Flow and Data Sharing. *J. Systems & Software*, 80(2):240–264, 2007.

[6] M. Abi-Antoun, J. Aldrich, N. Nahas, B. Schmerl, and D. Garlan. Differencing and Merging of Architectural Views. *Automated Software Eng.*, 15(8):35–74, 2008.

[7] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, pages 1–25, 2004.

[8] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *ICSE*, pages 187–197, 2002.

[9] A. Christl, R. Koschke, and M.-A. Storey. Equipping the Reflexion Method with Automated Clustering. In *WCRE*, 2005.

[10] D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, pages 48–64, 1998.

[11] P. Clements et al. *Documenting Software Architecture*. Addison-Wesley, 2003.

[12] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty Years of Graph Matching in Pattern Recognition. *Int. J. Pattern Recognit. Artif. Intell.*, 18(3):265–298, 2004.

[13] C. Flanagan and S. N. Freund. Dynamic Architecture Extraction. In *FATES-RV*, 2006.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[15] D. Garlan *et al.* The Acme Architectural Description Language. `http://www.cs.cmu.edu/~acme`.

[16] T. Hill, J. Noble, and J. Potter. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *J. Visual Lang. and Comput.*, 13(3):319–339, 2002.

[17] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *TSE*, 27(2), 2001.

[18] jRM. `http://jrmtool.sourceforge.net`, 2003.

[19] R. Koschke and D. Simon. Hierarchical Reflexion Models. In *WCRE*, 2003.

[20] P. Lam and M. Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *ECOOP*, pages 275–302, 2003.

[21] S. Lee, G. Murphy, T. Fritz, and M. Allen. How can diagramming tools help support programming activities. In *VL/HCC*, pages 246–249, 2008.

[22] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *TSE*, 21(9):717–734, 1995.

[23] K.-K. Ma and J. S. Foster. Inferring Aliasing and Encapsulation Properties for Java. In *OOPSLA*, 2007.

[24] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *FSE*, 1996.

[25] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *TSE*, 26(1), 2000.

[26] A. Milanova, A. Rountev, and B. G. Ryder. Constructing Precise Object Relation Diagrams. In *ICSM*, 2002.

[27] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized Object Sensitivity for Points-To Analysis for Java. *TOSEM*, 14(1):1–41, 2005.

[28] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *TSE*, 27(4):364–380, 2001.

[29] R. W. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, CMU, 2001.

[30] M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. In *POPL*, 1999.

[31] J. Schäfer and A. Poetzsch-Heffter. A Parameterized Type System for Simple Loose Ownership Domains. *Journal of Object Technology*, 5(6):71–100, 2007.

[32] J. Schäfer, M. Reitz, J.-M. Gaillourdet, and A. Poetzsch-Heffter. Linking Programs to Architectures: an Object-Oriented Hierarchical Software Model based on Boxes. In *Common Component Modeling Example (CoCoME)*, pages 238–266, 2008.

[33] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering Architectures from Running Systems. *TSE*, 32(7):454–466, 2006.

[34] M. Sefika, A. Sane, and R. H. Campbell. Monitoring Compliance of a Software System with its High-Level Design Models. In *ICSE*, pages 387–396, 1996.

[35] M.-A. Storey, C. Best, and J. Michaud. SHriMP Views: An Interactive Environment for Exploring Java Programs. In *IWPC*, page 111, 2001.

[36] P. Tonella and A. Potrich. *Reverse Engineering of Object Oriented Code*. Springer-Verlag, 2004.

[37] R. van Ommering, R. Krikhaar, and L. Feijs. Languages for Formalizing, Visualizing and Verifying Software Architectures. *Computer Languages*, 27(1-3):3–18, 2001.