Static Conformance Checking of Runtime Architectural Structure[1]

Marwan Abi-Antoun **Jonathan Aldrich**

September 2008
CMU-ISR-08-132

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

It is hard to statically check a system's conformance to its runtime architectural structure. Previous approaches address the code architecture, change the language radically, mandate implementation frameworks, or use dynamic analyses that cannot check all possible program runs.
We propose a static approach that supports existing object-oriented implementations, but relies on program annotations to encode architectural intent. We statically extract a hierarchical view of the runtime object graph from the annotated program and map it into an as-built runtime architecture. We then check and measure the structural conformance of the as-built and the as-designed architectures.
An evaluation on several systems showed that the approach can identify interesting structural non-conformities.

[1]This technical report supersedes the earlier technical report CMU-ISRI-07-119, entitled *Checking and Measuring the Architectural Structural Conformance of Object-Oriented Systems.*

This work was supported in part by NSF CAREER award CCF-0546550, DARPA contract HR00110710019, the Department of Defense, and the Software Industry Center at Carnegie Mellon University and its sponsors, especially the Alfred P. Sloan Foundation.

Keywords: runtime architecture, conformance checking, conformance measurement

Contents

List of Figures

1 Introduction

Many architectural views are needed to describe a software system. A *code architecture* or *module view* organizes code entities in terms of classes, packages, layers and modules. A system's *runtime architecture* or *runtime view* models runtime entities and their potential interactions [10].

Developers have a conceptual model of their architecture that is mostly accurate, but this model may be a simplification of reality [33, 8]. Thus, checking the conformance of the implementation to its desired architecture is a practically relevant problem [33, 8]. Previous research addressed the conformance of the code architecture [33, 35]. But checking the conformance of runtime architectures remains challenging for object-oriented systems, where a runtime view may bear little resemblance to a module view.

Some approaches delay conformance checking until runtime [37, 36], but a dynamic analysis can only check a few program runs. To meet certain quality attributes, even an atypical system execution must not introduce a critical architectural violation. A dynamic analysis cannot prove that a program always satisfies a particular property. That requires a static analysis, ideally, one that is *sound* and reveals all entities and relations that could possibly exist at runtime.

This paper describes an iterative multi-stage approach to statically check the conformance of a system to an architectural description. We extend the extract-abstract-check strategy [33, 12], and: (a) add annotations to the code to clarify the architectural intent; (b) extract a representation of the as-built runtime structure; (c) abstract a representation of the runtime architecture suitable for comparison; and (d) run an analysis to establish the key differences between the as-built and the as-designed architectures, both visually and through conformance metrics. The approach uncovered a number of unexpected facts about the implementation of several real representative object-oriented systems.

The approach builds on some of our previous work. First, we use a static analysis that extracts a hierarchical representation of the runtime object graph from an annotated program [4]. This paper abstracts that representation into a standard Component-and-Connector (C&C) runtime view [10]. Second, we adapt a structural comparison algorithm [6] to conformance checking. Unlike synchronizing two views to make them identical, conformance checking allows an as-built view to contain low-level details, and does not carry them over into the as-designed view. To preserve soundness, the analysis still accounts for communication in the as-built view that is not in the as-designed view.

This paper's contribution is a semi-automated static approach to check the conformance of a system's runtime architecture. The approach uses program annotations instead of radical language extensions [8] or implementation frameworks [29]. It relates a hierarchical object graph, that represents the instance structure of source code entities, to an as-designed hierarchical runtime view. This paper also introduces runtime architectural conformance metrics, qualified by annotation metrics.

Outline. The rest of this paper is organized as follows. Section 2 describes the overall strategy. Section 3 discusses abstracting an as-built runtime architecture. Section 4 discusses checking and measuring the structural conformance. In Section 5, we evaluate the approach. A discussion and related work (Section 7) round out the paper.

2 Positioning and Background

As a running example, we use Aphyds, an 8-KLOC Java system. Its original developer drew an as-designed architecture (Fig. 1). User interface components are in the upper half. A `circuit` and computational elements are the lower half. An Architecture Description Language (ADL) can model an informal boxes-and-lines diagram such as the above. In a runtime view, a *tier* or *group* is a *conceptual partitioning of functionality* [10]. For example, Aphyds follows a Document-View architecture and has two tiers.

Reflexion Models. Conformance checking relates source code entities to entities in a high-level model [33]. Reflexion Models (RM), a standard in conformance checking the code architecture [21], works as follows. In RM, a third-party tool extracts a *source model* from the source code. A developer posits an as-designed *high-level model* and a *map* between the source and high-level models. RM pushes each interaction described

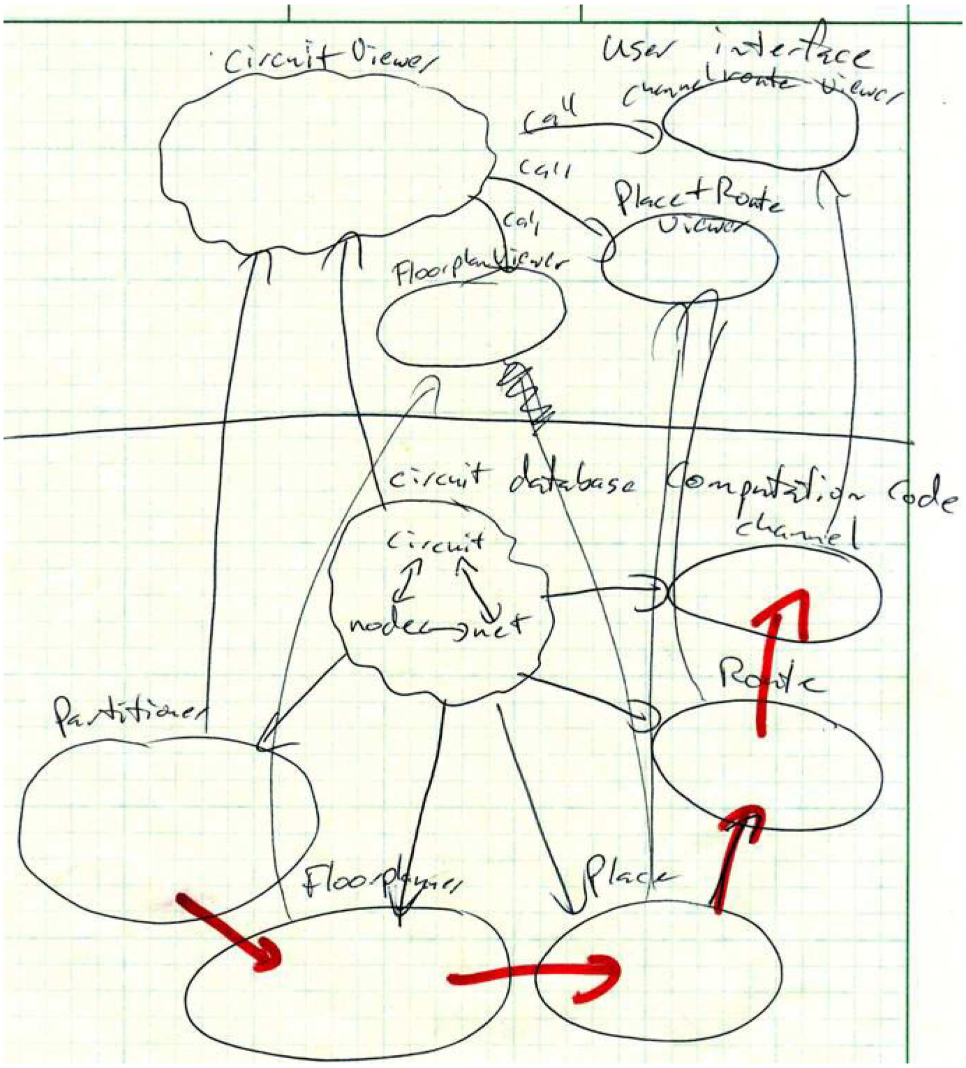


Figure 1: Aphyds as-designed architecture.

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 and shows the differences. A developer can iteratively: (a) modify the high-level model; (b) modify the source model; (c) modify the map; (d) trace a conformance finding to code; and (e) optionally, change the code to avoid an architectural violation.

The Java RM tool, jRM [17], can map class `Circuit` to a circuit high-level element in a code architecture. Assume briefly that RM supports tiers and qualifies an element by its tier using the :: symbol.
`class Circuit` to MODEL::circuit.

Let us hypothetically apply RM to runtime architectures to better understand the issues that would arise. For a runtime architecture, the source model must reflect the *runtime structure* of the source code entities. A class is not a runtime entity. Hypothetically, the above could indicate that all instances of the `Circuit` class map to `circuit`. But in an object-oriented system, there is usually more than one instance of any given class, and each instance can map to a different element in a runtime view.

Instead of mapping a class or all of its instances, we need to map runtime objects instead. But a static analysis knows only about field or variable declarations in a program, which denote references to runtime objects. For example, `Main.circuit`, a `circuit` field declared in class `Main`, points to an instance of the `Circuit` class at runtime. Hypothetically, assume we extend jRM and map:

`field Main.circuit` to MODEL::circuit.

This introduces the issue of the map's correctness. In an object-oriented system, multiple code elements could map to the same element in a runtime view. A reference of type `Circuit` and one of type `ICircuit`, an interface that `Circuit` implements, could alias and point to the same object at runtime. It would be incorrect to map the same runtime object to multiple high-level elements. Somehow, we must map all the references that *may* alias (i.e., refer to the same object), to the same element in the runtime view. In addition, one must be able to map one code entity to multiple elements in a runtime view. To support this feature, a previous system defines a context parameter using an annotation in the code, and binds that parameter to different actual contexts using additional annotations [24]. But that system supports neither inheritance nor hierarchy.

ADLs support the hierarchical decomposition of a component into a nested sub-architecture [30]. For example, the Aphyds as-designed architecture shows node and net inside circuit's sub-structure (Fig. 1). We would like to model secondary `node` and `net` objects as *part of* the primary `circuit` object. So we create a DB tier inside circuit, and map a field `node` to an element nested in `DB`:
`field Circuit.node` to MODEL::circuit.DB::node.

In summary, this paper generalizes previous maps [33, 24], accounts for inheritance and aliasing, and relates a hierarchical high-level model to a hierarchical representation of the runtime structure of source code entities.

2.1 Conformance Strategy.

We extend the extract-abstract-check strategy [33, 12]. The steps are:

1. Document the as-designed runtime architecture;
2. Abstract an as-built view from the implementation:
   (a) Add annotations to the code;
   (b) Extract a hierarchical representation of the runtime object graph, ideally, one that is sound;
   (c) Abstract that representation into an as-built view;
3. Structurally compare the as-built and the as-designed architectures to check their conformance; and
4. Compute a measure of their structural conformance.

In the terminology of Murphy et al. [33], the conformance check identifies:

Convergence: a node or an edge that is both in the as-built view and in the as-designed view;
Divergence: a node or an edge that is in the as-built view but not in the as-designed view;
Absence: a node or an edge that is in the as-designed view but not in the as-built view.

2.2 Abstraction Strategy

At runtime, an object-oriented program can be represented as an *object graph*: nodes correspond to objects, and edges correspond to relations between objects. Previous static analyses that do not use annotations produce low-level non-hierarchical object graphs [16]. Such representations explain runtime interactions in detail but convey little architectural abstraction ((Fig.2).

We follow Reflexion Models and avoid reverse engineering abstractions that architects do not recognize [33]. In our approach, a developer guides the architectural abstraction by adding annotations to clarify the architectural intent in the code. The annotations specify object encapsulation, logical containment and tiers, which are not explicit constructs in general purpose programming languages. These annotations support abstract reasoning about data sharing and make the extracted architecture reflect an architect's design intent rather than a tool's heuristic.

The annotations specify and enforce the sharing of state between objects, a key challenge in extracting a runtime architecture. This state sharing is often not explicit in object-oriented programs but instead, is implicit in the structure of references created at runtime.

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

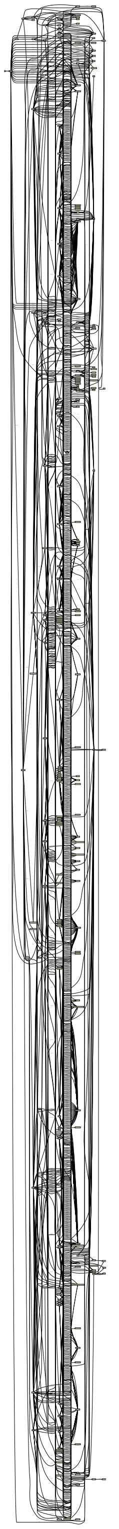


Figure 2: Thumbnail of the Aphyds object graph obtained at compile-time by WOMBLE [16]. Some node labels become semi-readable after zooming in by 6400%.

```
1  class Circuit {
2    public domain DB; // Declare public domain DB
3    domain OWNED; // Declare private domain OWNED
4    DB Node node; // Declare Node reference in DB
5    // Outer annotation is for container; inner one for its elements
6    OWNED Hashtable<String, DB Node> nodes;
7  }
8  class Viewer<M> { // Declare domain parameter M
9    M Circuit circuit; // Declare Circuit reference in M
10 }
11 class Main {
12   domain MODEL, UI; // Declare top-level domains
13   MODEL Circuit circuit;
14   // Bind domain parameter M to actual domain MODEL
15   UI Viewer<MODEL> viewer;
16 }
```

Figure 3: Partial Aphyds annotations.

that govern how it can reference objects in other domains [7].

Fig. 3 shows the annotations that a developer might add to some Aphyds classes. The actual annotation system uses existing language support for annotations. But here, we use a simplified syntax that extends the language. A developer indicates what domain an object is part of by annotating each reference to that object in the program (Lines 4,6). One typically chooses domain names to convey architectural intent. We use capital domain names to distinguish them from other program identifiers.

Ownership domains may be declared at the top level of the application (Line 12) or within an object (Lines 2,3). Each object can declare one or more *public* or *private* domains to hold its internal objects. The domains within an object express a sub-structure within the object, one that consists of other domains and objects that represent its parts. Objects inside a private domain such as `OWNED` are *encapsulated*. But having access to a `Circuit` object gives the ability to access instances of `Node` and `Net` inside its public domain `DB`. An instance of the `Viewer` class accesses other objects in the `MODEL` domain, by declaring a formal *domain parameter* `M` on the `Viewer` class, and *binding* that parameter to domain `MODEL` (Line 15).

Although a domain is declared inside a class, each instance of that class has its own runtime domain, and whenever our analysis distinguishes two objects, it also distinguishes the domains they contain.

A typechecker validates the annotations, identifies where the annotations are inconsistent, or where the code violates an annotation. For instance, one cannot have a `public` method that returns an alias to an object inside a private domain. This instance encapsulation is stronger than making a field `private` to restrict its module visibility.

Object Graph. A static analysis scans the annotated program's abstract syntax tree and produces a sound hierarchical representation of all possible runtime object graphs, the Ownership Object Graph (OOG). Compared to non-hierarchical object graphs, an OOG provides abstraction by ownership hierarchy and by types [4].

Collapsing many nodes into one is a classic approach to shrink a graph. However, an OOG collapses nodes based on the actual runtime and ownership structure, not according to where objects were declared in the program, some naming convention or a graph clustering algorithm.

An OOG is a graph with two node types, domains and objects. The nodes form a hierarchy where each object node has a unique parent domain, and each domain node has a unique parent object. Edges between objects correspond to field references or usage relations. The OOG uses a nested-box visualization. Dashed-border white-filled boxes represent domains, and grey-filled boxes represent objects. A private domain has a thicker dashed border. Field references are shown as solid edges. We label each object with one of its types. A (+) symbol indicates an elided sub-structure.

Example. One Aphyds OOG shows two top-level domains, `MODEL` and `UI`, and objects `Circuit` and `Viewer` in those domains (Fig. 4(a)). In an OOG, a `Circuit` refers to a runtime entity, not the class `Circuit`, even when we refer to an object by its type.

Key Properties. We highlight several properties of the OOG and their relevance to conformance checking.

Object Abstraction. Different executions may generate a different number of objects. The OOG summarizes multiple runtime objects with a canonical object. For instance, there could be many `Node` objects at runtime, but only one appears in the `DB` domain. In a runtime view, one component can represent many instances at runtime.

The ownership domains type system guarantees that two references in different domains can never alias. If two variables *within the same domain* may refer to the same object at runtime, the OOG shows them as one. A runtime architecture would be deceptive if it showed one runtime entity as two components. For instance, an architectural security analysis could assign one runtime entity two different values for a key trustLevel property.

Object Lifting. The OOG transitively lifts objects from formal domains into actual domains, to show all the objects in a given domain, not only those that were declared directly in that domain. Indeed, the `Circuit` object in `Viewer`'s domain parameter `M` is the same as the `Circuit` object in the `MODEL` tier. Lifting allows mapping multiple code elements to one runtime element. By binding one domain parameter to different actual domains, lifting can also map one code element to multiple design elements.

Hierarchy. Conformance checking must distinguish between objects that are architecturally irrelevant, and objects that exist in the implementation but are missing from the as-designed view. Typically, in an OOG, only primary objects appear in the top-level domains. And each of those objects have more domains and objects, until low-level objects are reached. For example, a Circuit object encompasses other objects that are of type Node and Net. Hierarchy enables varying the abstraction level [38], to obtain an as-built view that a tool can structurally compare to an-designed architecture.

Edge Lifting. Reflexion Models shows an edge from `Viewer` to `FloorPlanUI`, only if `Viewer` declares a reference to a `FloorPlanUI` object. In an OOG, a `Viewer` references a `Displayer`, and `Displayer` references a `FloorPlanUI`. `Viewer` holds `Displayer` in a public domain since it is not a primary object. When `Viewer`'s sub-structure is elided, the OOG lifts that relation to `Viewer`, and shows a *summary edge* from `Viewer` to `FloorPlanUI`, shown as a dotted edge in the OOG (Figs. 4(a), 4(b)).

Traceability. Each OOG element can be traced to a set of nodes from the program's abstract syntax tree.

Soundness. The OOG is proven sound [4].

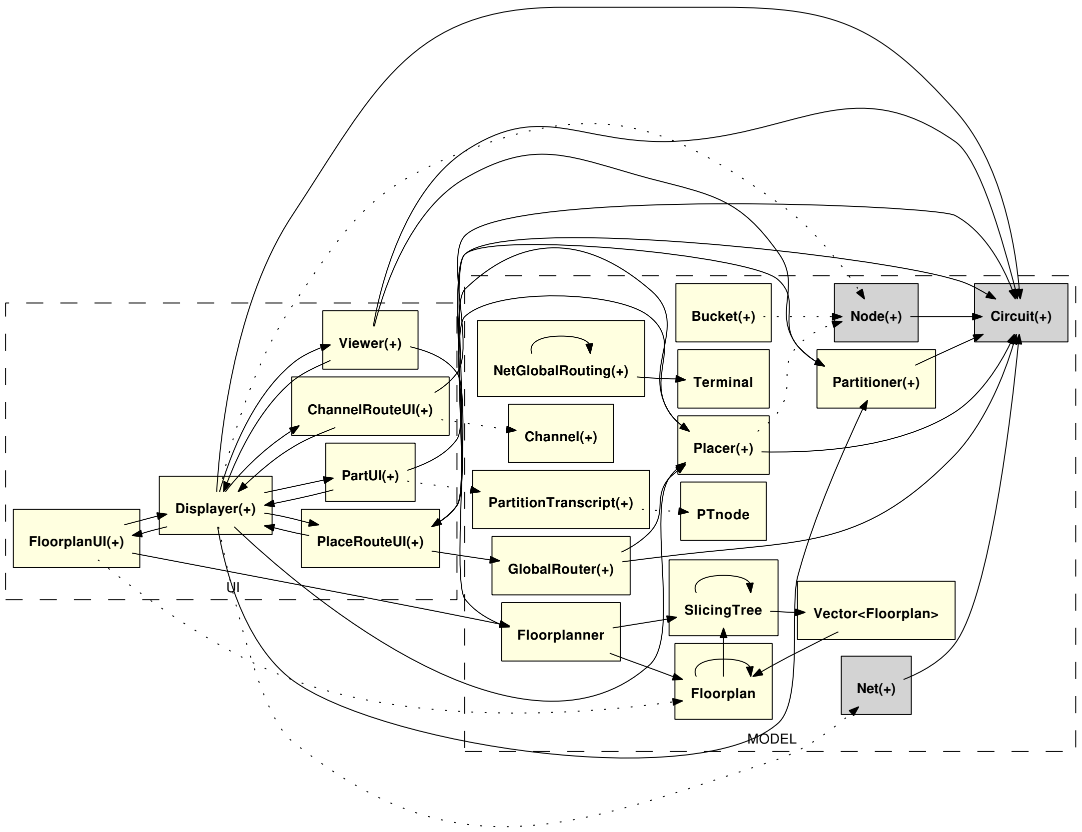
3 Architectural Abstraction

C&C Definitions. We document the as-designed C&C view in the Acme general purpose ADL [14]. Most other ADLs also support the following elements [30]. A Component is a unit of computation and state. A Port is a point of interaction on a Component. A Connector represents an interaction among Components. A System is a configuration of Components and Connectors. Acme also supports the hierarchical decomposition of a Component 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.

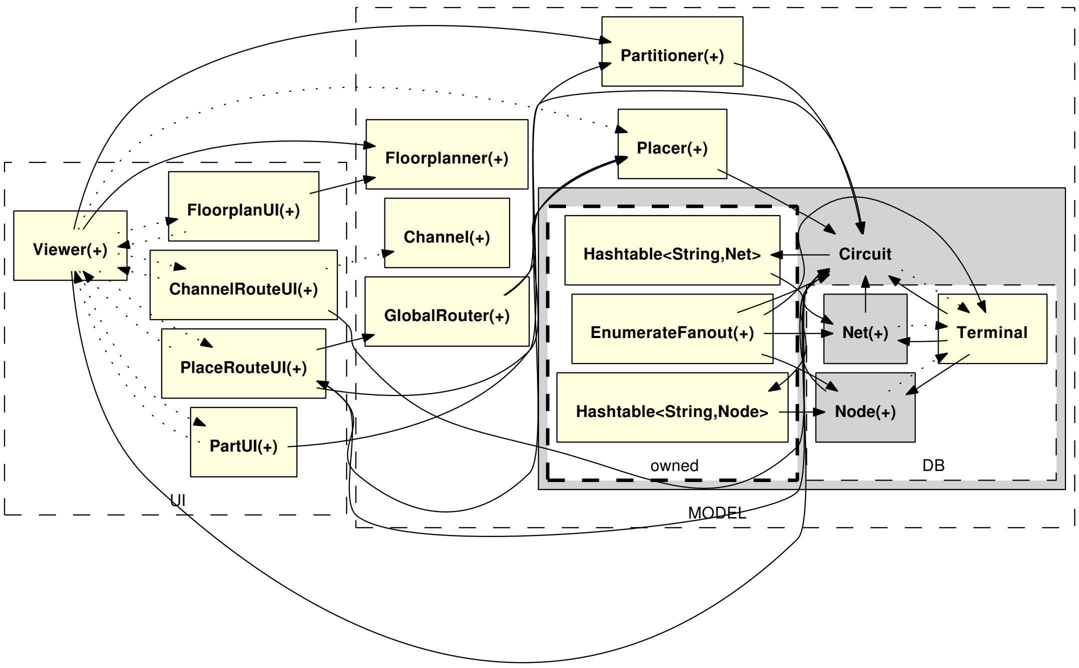
Architectural Types. We enrich the base architectural model with types and properties, which improve the precision of the structural comparison [6]. Two generic Component and Connector types, CompT and ConnT, respectively, hold properties for the conformance analysis.

A Port that provides services has type ProvideT, and a Port that requires services has type UseT. The structural comparison uses the type information to avoid matching a ProvideT Port to a UseT Port, for example.

Mapping an OOG to a C&C View. We represent the as-designed architecture as a C&C view. We also convert an OOG into an-built C&C view for comparison with the as-designed view. We later refine



(a) Aphyds OOG, with annotation defining top-level and private domains.



(b) Aphyds OOG, after defining additional public domains.

Figure 4: Two possible Aphyds OOGs.

the conversion to generate a more abstracted as-built C&C view, but for now, the base conversion works as follows.

The root object of an OOG is often an instance of a class that only declares the top-level domains and objects inside them. So the root object maps to a System, and the top-level domains map to the top-level tiers in the System.

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

References between objects create Ports as follows, while excluding uninteresting self-edges. 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.

Structural Constraints. To be structurally comparable, both the as-built and the as-designed views follow similar conventions. In Acme, a Component can be included in more than one Group. But in ownership domains, an 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. If Connector c connects two Components that are in the same Group g, c must be also in g.

Port Directionality. An Acme Port has no built-in directionality. Its type specifies whether it provides services (ProvideT) or uses services (UseT). In some cases, the as-designed view may have a Connector between two Components, but the connection in the as-built view may be in the reverse direction. The conformance check 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. For the above example, the analysis shows a Connector, and ProvideT and UseT Ports, for the communication in the opposite direction. Such a stylized use of ports also seems easier to understand [8].

Other conventions are possible, as long as both the as-designed and as-built views adopt them. For example, to reduce clutter, one could have a single ProvideT Port and a single UseT Port per Component.

Abstraction and Conformance. An as-designed view is often an abstracted view of the system's architecture. However, to avoid misleading developers, an as-designed view must still represent all communication that could possibly exist in the as-built system. As long as a program is fully annotated and the annotations typecheck, its OOG is a sound approximation of its runtime structure. And by construction, so is the as-built C&C view.

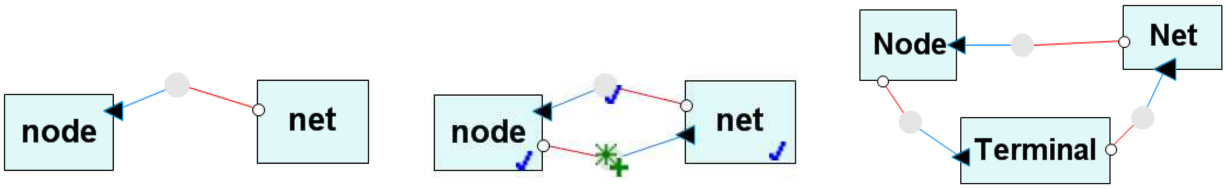
To enable structural comparison, the as-designed and the as-built views must have a similar number of top-level tiers, a similar hierarchical decomposition, and a similar number of components and tiers at each hierarchy level. One achieves the desired number of components in the as-built view by refining the annotations, controlling the OOG or adjusting its conversion to a C&C view.

One refines the annotations by pushing secondary objects underneath primary objects using the strict encapsulation of private domains, or the logical containment of public domains. One controls the OOG abstraction by ownership hierarchy and by types. We do not discuss here in detail abstraction by types in an OOG [4]. For instance, we did not use that feature for Aphyds.

Additional Abstraction. For comparison with a manually generated as-designed view, we control the conversion of an OOG into a C&C view, in the following ways:

Add types and properties. A developer optionally maps implementation types to architectural types, to get a richer as-built view with types and properties.

Elide private domains. A developer optionally elides private domains, which typically contain implementation details. For Aphyds, we elide the `OWNED` domain on `Circuit`, which stores `Hashtable`s of `Node` and `Net` objects (Fig. 4(b)). The conversion analysis adds summary edges to account for communication through elided objects.



(a) As-designed C&C view. (b) Conformance view. (c) As-built C&C view.

Figure 5: Displaying conformance.

Elide singleton tiers. 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 a single DB tier inside circuit would be counterintuitive. Unless the developer requests otherwise, the conversion does not create a singleton tier inside a Component. Unlike eliding private domains, the conversion still creates sub-components for the objects in those unmapped domains. When skipping the private `OWNED` domain inside `Circuit`, the conversion creates node and net directly inside circuit, and does not create a DB tier (Fig. 5.3).

Merge objects. Detecting split or merged nodes is difficult and currently unsupported [6]. A developer optionally merges selected objects in the OOG into one component, and the conversion merges the corresponding edges.

Skip objects beyond a certain depth. The mapping converts the OOG up to a user-selected depth, typically the hierarchical decomposition depth in the as-designed view. Reducing the as-built view size speeds up the comparison. Restricting the hierarchy depth does not affect conformance, because summary edges account for the elided substructures in a depth-restricted OOG.

4 Architectural Conformance

A system conforms to its as-designed architecture if the latter is a conservative abstraction of the system's runtime structure. The *communication integrity* principle stipulates that *each component in the implementation may only communicate directly with the components to which it is connected in the architecture* [32, 27].

Assumptions. We compare the as-designed and the as-built architectures using a structural comparison that works with hierarchical views, does not assume unique identifiers, detects renames and restricted moves and allows forcing or preventing matches [6]. These assumptions closely match the problems of post-hoc view synchronization and conformance checking.

Displaying Conformance. Unlike view synchronization, conformance checking does not actually modify the as-designed view. Instead, the conformance analysis produces a *conformance view* of the as-designed architecture, which shows convergences and absences graphically. It represents divergences by showing additional Connectors that are present in the as-built view but are missing from the as-designed view.

Conformance Properties. All elements have a finding property, set to Convergent(✓), Divergent (+) or Absent (✗). An element's visual display is based on its properties.

All elements have a traceability property, a set of filename and line number pairs. The conversion of an OOG to an as-built C&C view sets each element's traceability property. For an element with a Convergent or Divergent value for its finding property, the analysis sets its traceability property in the conformance view based on its matching element in the as-built view. Based on this information, a developer can trace directly to the pertinent lines of code, and not have to potentially review the entire code base.

4.1 Checking Conformance

In conformance checking, the as-designed view is more authoritative than the as-built one. First, it is an architect's abstracted view, in that the components she included may be more relevant than those she omitted. Second, she may have chosen names that convey her architectural intent. The analysis works as follows:

Highlight differing connections: the analysis matches the components in the as-built view to those in the as-designed view, and shows differing connections as divergences or absences. For instance, if the as-built view has a connector between `FloorPlanUI` and `Viewer`, and the latter match the as-designed components floorplanUI and viewerUI, the analysis shows as divergences the additional Ports and Connectors between floorplanUI and viewerUI.

Use as-designed view names: element names in the as-designed and the as-built views may not match exactly. The structural comparison can detect renames [6]. E.g., the check correctly matches as-built `Viewer` component to as-designed component viewerUI. Unlike view synchronization, however, the conformance analysis does not propagate the as-built view names to the as-designed view. For example, it matches floorplanUI to `FloorPlanUI` and viewerUI to `Viewer`, and shows communication between floorplanUI and viewerUI, without renaming them.

Summarize divergent components: If there are components in the as-built that are not in the as-designed view, the analysis does not directly add them (Fig. 5), to avoid cluttering the as-designed view with low-level implementation details. To enforce communication integrity, however, the analysis still accounts for any communication in the as-built view that is not in the as-designed view, including communication through these unmapped components.

In an as-built view, `Node` connects to `Terminal` and `Terminal` to `Net` (Fig. 5(c)). The as-designed view has node and net, but has no component matching `Terminal` (Fig. 5(a)). The analysis matches node to `Node`, and net to `Net`, respectively. It then shows a Divergent Connector from node to net in the as-designed view, since one does not already exist (Fig. 5(b)). For emphasis, we decorate a summary Connector, which can be either Divergent or Convergent, with the ✳ symbol.

Viewed differently, a summary Connector represents any objects in the as-built view that do not have counterparts in the as-designed view. This allows an as-designed view to have a coarser granularity of components, and optionally abstract multiple interacting objects with a connector.

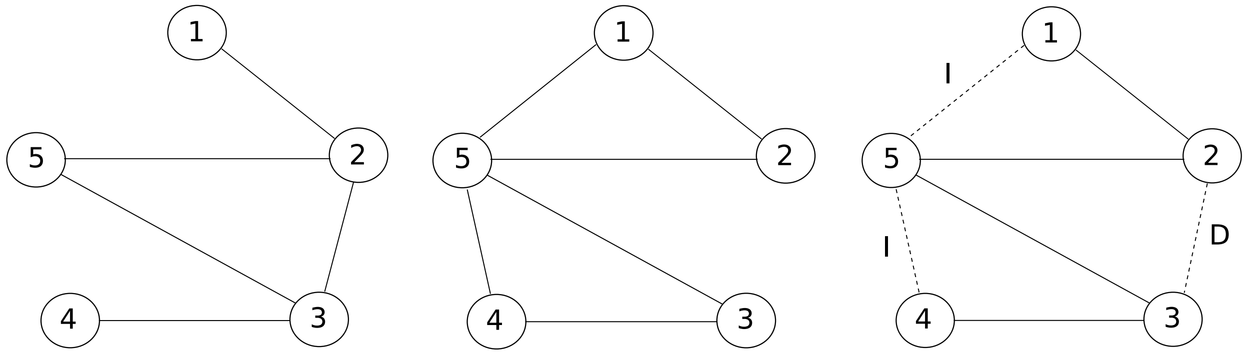
Check matching sub-structures: As-designed views are often hierarchical. An OOG provides abstraction primarily through ownership hierarchy by folding low-level objects into higher-level component instances. Typically, as-built components will have more detailed sub-structures than their as-designed counterparts. But the conformance check does not analyze the additional substructures, to avoid generating many false positives. This preserves soundness because both the OOG and the as-built C&C view have summary connectors that represent any communication through any elided or skipped substructures.

For instance, the as-designed viewerUI does not define a substructure. So the analysis ignores any substructure in the corresponding as-built `Viewer`. But the as-designed circuit has substructure and matches the as-built `Circuit`. In that case, the analysis recursively checks the substructures of circuit and `Circuit`. When converting an OOG into a C&C view, we excluded private domains, such as `Circuit`'s `OWNED`. Since `OWNED` is at the same level as `DB`, the analysis would have otherwise processed `OWNED` and generated undesired divergences.

For each metric, we define: (a) *base measures*; (b) any *derived measures*; (c) *indicators* on how to interpret those measures; and (d) the overall value or risk [28].

4.2 Measuring Conformance

Similarly to Reflexion Models (RM), we count edge convergences (CE), divergences (DE) and absences (AE). However, in RM, if the map generates a node that is not the as-designed view, RM automatically adds it to the as-designed view. As a result, RM has no divergent or absent nodes, and does not compute summary edges. We count summary edges (SE), as well as node convergences (CN), divergences (DN), and absences



(a) Graph g_A. (b) Graph g_B. (c) Editing g_A.

Figure 6: Graph edit distance (GED).

(AN). In our approach, AN and DN indicate that the as-designed view omits components from the as-built view, or uses a different system decomposition (Table 1).

To explain the source of the divergences and summary edges, the conformance check can optionally show each Divergent Component in the conformance view, but does not show Connectors to such divergent components.

To get one conformance metric, we combine edge divergences and absences, using the notion of *graph edit distance* (GED) (Fig. 6). GED models inconsistencies by transforming one graph into another [11]. Typical edit operations include the deletion, insertion and relabeling of nodes and edges. Each edit operation is assigned an application-dependent cost. Here, renames have a zero cost. The edit distance of two graphs g_A and g_B is the sequence of edit operations with the minimum cost that transform g_A into g_B. Fig. 6(c) transforms g_A into g_B with 2 edge insertions (I) and 1 edge deletion (D) in dotted lines, i.e., $GED = 3$.

Computing the GED is NP-hard, except for graphs with unique node labels [11], which is not the case for architectural views. Having a mapping between the nodes in the two graphs, which we obtain using structural comparison, is like having unique labels.

CCM. The Core Conformance Metric (CCM) counts the number of unique edge deletions (D) and insertions (I), i.e., the absences (AE) and divergences (DE), respectively, that would make the as-built view account for all communication in the as-designed view. To get a percentage, we divide by the total number of edges and subtract from 100%. It is better to have fewer absences and divergences. So a higher CCM value indicates a higher structural conformance.

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

Annotation Metrics. Reflexion Models tracks unmapped entries in the source model. Similarly, we qualify our conformance metrics with measurements of the program annotations. To measure the percentage of the program that lacks annotations, we use a derived measure, WARN, namely the number of annotation warnings that the annotation typechecker generates. Except for some defaults for immutable objects, such as those of type `String`, 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). The metric WARN% normalizes WARN by the number of object references the program declares. WARN% is an indicator of how many annotations are missing to make the OOG soundly represent the as-built runtime architecture. A lower WARN% is better. For a program without annotations, WARN% will be high. As valid annotations are added, WARN% decreases. For Aphyds, WARN% is 5%.

All the subject systems we studied still have warnings that require refactoring the code or increasing the type system's expressiveness. We manually examined the warnings and believe they do not contribute to missed architectural violations.

5 Evaluation

The evaluation aimed to answer the research question: *Can the approach identify interesting architectural structural non-conformities in real object-oriented systems?*

5.1 Tool Support

We developed several Eclipse plugins to relate C&C views, OOGs and Java source files:

AcmeStudio is an Acme modeling environment [14];

ArchDomJ typechecks the annotations;

ArchRecJ recovers an OOG from annotated code;

ArchCog maps an OOG to a C&C view (Section 3);

ArchConf checks the conformance between two C&C views and computes the metrics (Section 4);

CodeTraceJ loads the traceability of an element selected in a C&C view, opens the corresponding Java files, and highlights the appropriate lines of code;

ArchMod modifies the original as-designed view to actually add a selected Connector marked as Divergent, or delete a Connector marked as Absent.

5.2 Methodology

Typically, the developer iterates one or more steps in the process. In Eclipse, the developer uses the AcmeStudio perspective to build the as-designed architecture. She then switches to the Java development perspective, loads the implementation project, adds ownership domain annotations to the code as Java 1.5 annotations, and invokes ArchDomJ. She double-clicks on a warning in the Eclipse problem window to go to the offending line of code. Once the program is mostly annotated, she uses ArchRecJ to extract an OOG, and refines the annotations until the OOG has a number of top-level components roughly comparable to that in the as-designed view.

The developer then invokes ArchCog to convert the extracted OOG into an as-built C&C view. ArchCog allows her to map implementation types to architectural types, elide private domains and restrict the projection depth. She then points ArchConf to the as-built and as-designed views, and examines the comparison results.

In ArchConf, the developer accepts the comparison results or manually forces or prevents matches between the two views and reruns the comparison. ArchConf then creates a conformance view of the as-designed architecture, and displays the conformance metrics in an output window. In the AcmeStudio perspective, the developer examines the conformance view, and investigates unexpected divergences. She uses CodeTraceJ to confirm a convergence or trace a divergence to the code. If the divergence is critical, she may modify the implementation to eliminate the architectural violation.

ArchConf does not modify the original as-designed view, but creates a conformance view as a copy. A developer can use ArchMod to commit a selected divergence or absence to the as-designed view.

In some cases, it may be difficult to find a documented as-designed runtime architecture for an existing system. In that case, getting an abstracted as-built C&C view may have value in itself. If a manually generated as-designed view is missing or out-of-date, the developer could take an as-built C&C view, elide unwanted detail from it, and treat it as an as-designed view.

Experience Overview We evaluated the end-to-end approach on three representative real systems totaling 38 KLOC. The study's subject (one of us, hereafter "we") developed several of the tools, but none of the subject systems. We iteratively annotated the code, extracted as-built views and related them to diagrams drawn by the original developers. The as-designed architectures we studied omitted connections, components or entire subsystems.

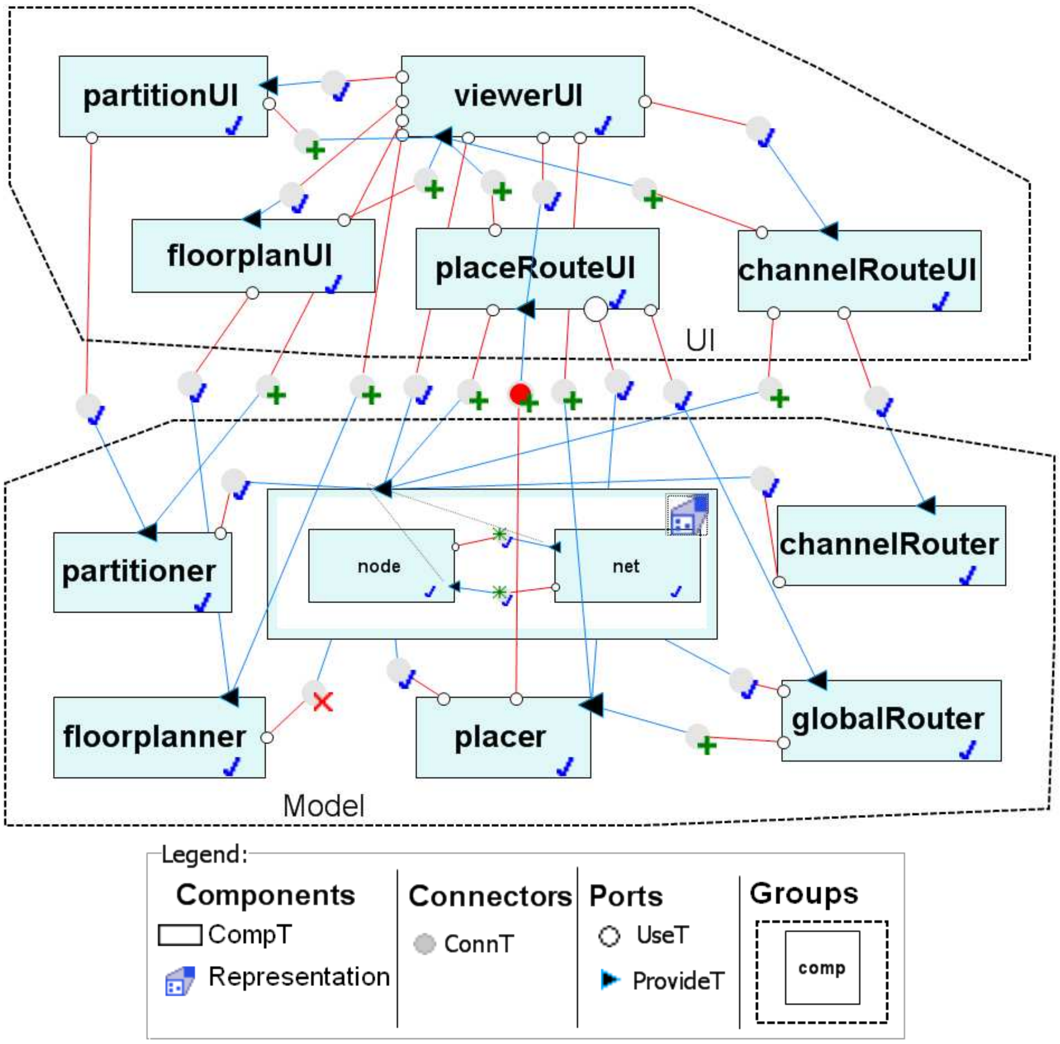


Figure 7: Aphyds conformance results.

5.3 Extended Example: Aphyds

We now complete the evaluation of the Aphyds system.

As-Designed Architecture. We based the Aphyds as-designed architecture on the informal diagram (Fig. 1), but iterated it a few times. We initially forgot to add a Connector joining two Components inside a Group g to that same Group g. This resulted in badly matched connectors. We also forgot to reverse the direction of arrows in the developer's diagram when adding some connectors [8, p. 192]. Because the structural comparison does not detect splitting or merging, we represented one component labeled "Place + Route" in the informal drawing (Fig. 1) as two separate components, placeRouteUI and channelRouteUI.

Annotations. We also iterated the program annotations, the OOG extraction, the conversion to a C&C view, and the conformance check. Table 1 shows how the conformance metrics evolved between the two iterations.

Iteration 1. We initially organized the Aphyds objects into two top-level domains. `UI` holds a `Viewer` 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 `Hashtable` objects inside a `Circuit` object.

These annotations produce an OOG that is hierarchical, as the (+) sign indicates (Fig. 4(a)). But that OOG still has many objects in the top-level domains, and so does the C&C view that ArchCog generates from that OOG suffers.

Conformance Metrics. The conformance check does not produce good conformance metrics (Table 1). For example, because `Circuit`, `Node` and `Net` are at the same level in the as-built C&C view, the conformance check marks as Absent the node and net components inside circuit (2 node absences).

The as-built C&C view has many more components in the top-level tiers than the as-designed view, hence the high node divergences. Moreover, the conformance analysis generates many summary connectors to account for possible transitive communication, and this accounts for the high number of edge divergences. 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`. Instead, since the analysis lacks information about logical containment, it shows a divergent summary connector from `Displayer` to `Placer`, and many others (97 in total). This almost turns the conformance view into a fully-connected graph and makes it unreadable.

Iteration 2. Using the as-designed architecture as a guide (Fig. 1), we defined several public domains. Some of these domains contain objects that we are trying to hide from the top-level tiers. For example, `Viewer` has a `DISPLAY` public domain to hold a `Displayer` object that other `UI` objects, such as `FloorPlanUI`, reference. But `Displayer` is not in the developer's diagram (Fig. 1).

Public domains can also abstract low-level objects into higher-level components. For example, `Circuit` holds objects such as `Node` and `Net` inside its `DB` public domain, to reflect the as-designed architecture (Fig. 1). Refining the annotations to define public domains required mostly local and incremental changes to the annotations.

With the revised 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. 4(b)). This OOG has a system decomposition close to one in the desired architecture (Fig. 1), and by construction, so does the as-built C&C view.

Conformance Check. 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. 5.3). In the as-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 ✳ (the earlier explanation and example of summary connectors in Fig. 5 were in fact adapted from this part of Aphyds).

Study Findings. The as-designed architecture (Fig. 1) is only about 60% accurate, based on the CCM metric. Indeed, there are several divergences between viewerUI and other UI components, between UI and MODEL components, and among MODEL components. Many uni-directional arrows turned out to be bi-directional in reality. A developer could use ArchMod to update the as-designed architecture with some of the divergent connectors.

One divergence that crosses tiers, from placer in MODEL to placeRouteUI in UI, was a red flag (the dark colored connector in Fig. 5.3). As a multi-threaded application, Aphyds 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 inspected the code to make sure that it handled the callback correctly.

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%

Performance. For Aphyds, the OOG extraction takes around 10 seconds, and the structural comparison takes between 57 seconds (Iteration 1) and 33 seconds (Iteration 2). We measured these times on an Intel Core 2 Quad Processor (2.40GHz) with 4GB of RAM running Windows XP.

Discussion. Aldrich et al. previously studied Aphyds, and identified similar architectural violations, but only after they re-engineered it to ArchJava [8]. They manually defined, in code, over 20 `component class`es and over 80 `port`s, refactored the program to neither take as argument nor return any reference to an instance of a `component class`, and inadvertently injected several defects [8].

In a C&C view extracted from ArchJava, `Component` a appears inside `Component` b if a instantiates b as one of its fields. So it may not be enough to simply convert a Java `class` into an ArchJava `component class`: one may need additional `component class`es just to capture the intended system decomposition. Using our annotation-based approach, one can achieve the desired decomposition with private and public ownership domains, and without defining additional classes. During our Aphyds evaluation, we just added annotations to the original Java program, without refactoring it, and followed the rest of the approach.

Adding ownership annotations to an existing system is less invasive than re-engineering it to ArchJava to expose its architecture [5]. The annotations, unlike ArchJava, do not change the system's runtime semantics. The annotations also support common object-oriented idioms, including passing references to objects. Moreover, 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 3,500 lines of Aphyds in 4 hours, a quarter of the time they spent re-engineering Aphyds to ArchJava [8].

We also conducted a field study to more reliably estimate the annotation effort. The first author spent 35 hours adding annotations to extract an OOG from a 30-KLOC module of a 250-KLOC system (WARN is still high, however). The architects did not provide us with an as-designed runtime architecture, however, so we could not check its conformance [3]. We could not have re-engineered that module to ArchJava in the same few days. So, for existing systems, annotations seem more adoptable than language extensions.

5.4 Extended Example: JHotDraw

JHotDraw (Version 5.3) has around 200 classes and 15,000 lines of Java. JHotDraw is a significant example in the object-oriented community, that is open source, rich with design patterns, uses both composition and inheritance heavily, and has evolved through several versions.

As-Designed Architecture. As is the case for many legacy systems, we were unable to find a documented execution architecture for JHotDraw. We did find however a documented abstracted code architecture [2, Fig.6]. Of course, the execution architecture may be significantly different from the code architecture. We used the code architecture as an estimate to be refined by the conformance checking step. For each class in the code architecture, we created a component instance. Then, for each association in the class diagram, we created a connection between the corresponding components.

JHotDraw's architecture posed another challenge. From a previous study, we knew that a `Drawing` was actually implemented as a `Figure`, contrary to the as-designed code architecture [1]. So the OOG, and thus by transformation, the as-built view, represented both `Drawing` and `Figure` with one runtime component. Had we modeled `Drawing` and `Figure` as separate in the as-designed view, the structural comparison would not have detected the splitting or merging [6]. This led us to merge `Drawing` and `Figure` into one `DrawingFigure` component in the as-designed view.

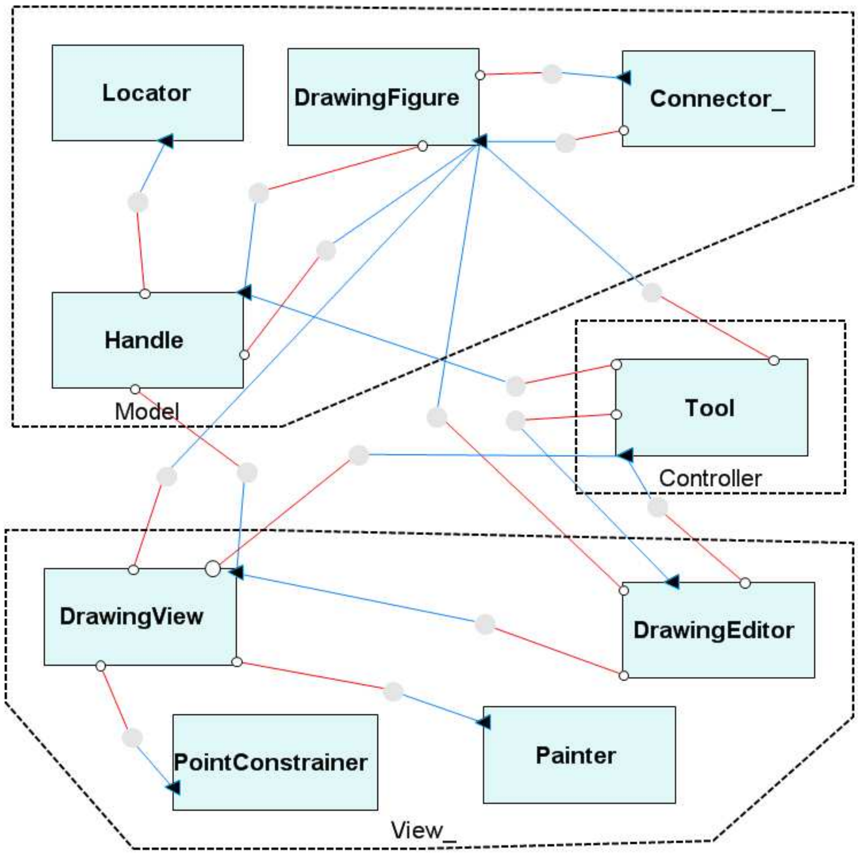


Figure 8: JHotDraw as-designed architecture documented in Acme.

Annotation Process. An often cited article discusses how JHotDraw follows the Model-View-Controller design pattern [18]. The JHotDraw package structure does not reveal that fact, since all the core types are in the same package. We added annotations to JHotDraw without refactoring, and organized objects into the following domains [2]:

`Model`: consists of `Drawing`, `Figure`, `Handle` objects, etc. A `Drawing` is composed of `Figure`s. A `Figure` has `Handle`s for user interactions;

`View`: consists of `DrawingEditor`, `DrawingView`, etc.;

`Controller`: `Tool`, `Command` and `Undoable` objects. `DrawingView` uses a `Tool` to manipulate the `Drawing`.

Results. ArchConf detected many renames and a few missing components, such as an `Undoable` in `Controller` and an `UndoManager` in `Model` Many connections, we thought to be unidirectional, such as between components `DrawingView` and `DrawingEditor`, turned out to be bi-directional (Fig. 10).

There was however one big surprise: there were no callbacks from `Model` into `Controller`! In the base MVC pattern, a controller registers itself with the model and receives notifications. Since there is no controller component, we suspected that the view acts also as controller, a common implementation optimization. Indeed, in the JHotDraw "CRC Cards View", the designers mention that `DrawingView` "handles input events" [13, Slide #10], a controller responsibility.

We looked more closely at the as-built C&C view and noticed a connection between `Handle` in `Model` and `Undoable` in `Controller`. But since `Undoable` did not connect to `Tool`, the conformance check did not add a summary connector between `Handle` and `Tool`. This example justifies the need for richer conformance

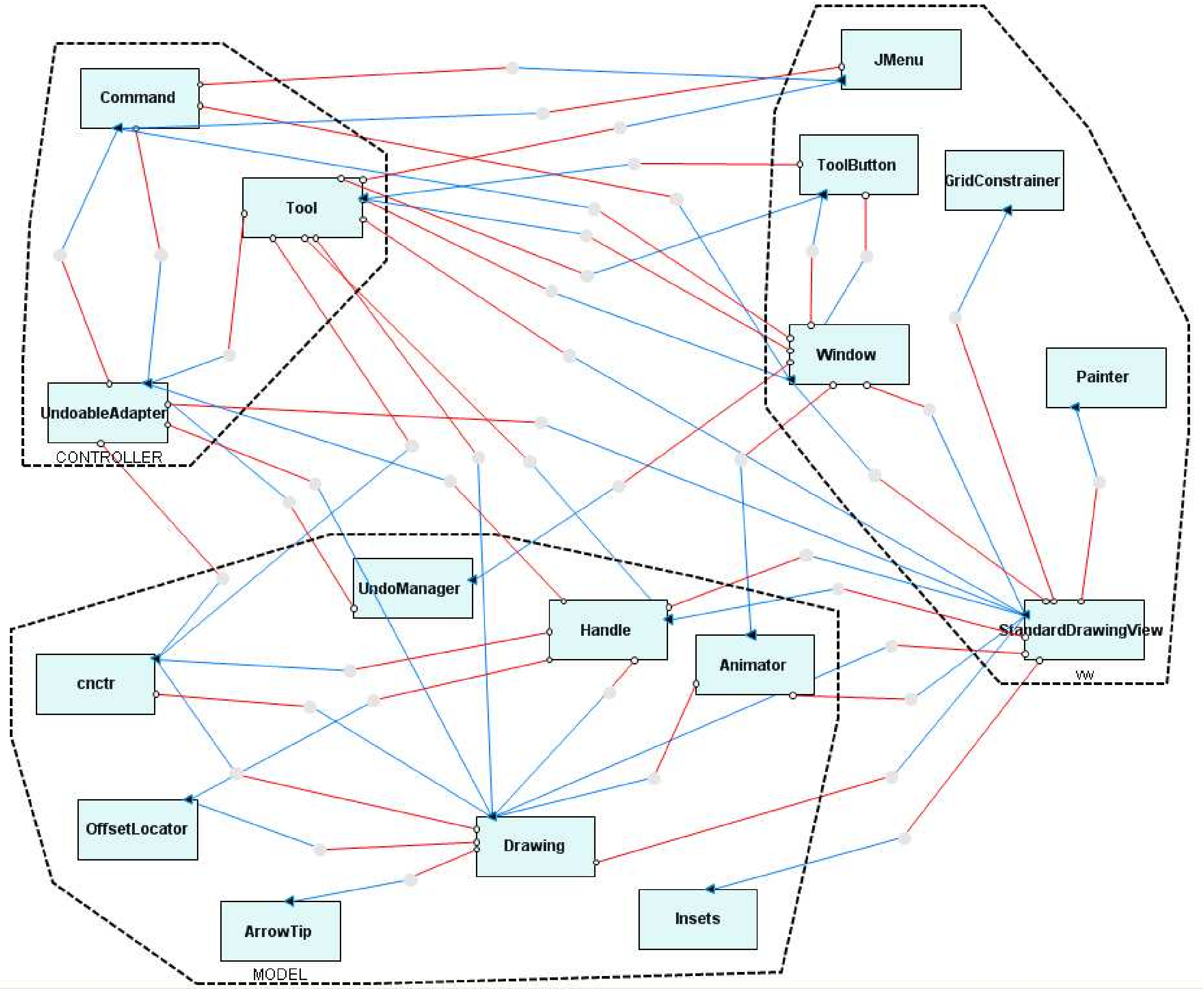


Figure 9: JHotDraw as-built architecture documented in Acme.

Table 2: JHotDraw conformance metrics.

System	CN	DN	AN	CE	DE	AE	SE	CCM
JHotDraw	9	8	0	23	49	0	72	32%
JHotDraw (no summaries)	9	8	0	16	7	0	0	70%

metrics that reflect the entire as-built view and not just divergences.

In fact, the as-designed architecture focused on the *domain model* and ignored the *application model*, which includes `UndoManager` and `Undoable`. These components are a later addition, part of a somewhat independent subsystem to implement undo, not mentioned in the documentation.

Metrics. The low CCM indicates a large proportion of divergences and absences (Table 2). This was to be expected because of how we obtained the as-designed view. Moreover, the as-designed view is missing several top-level components, in each of the tiers.

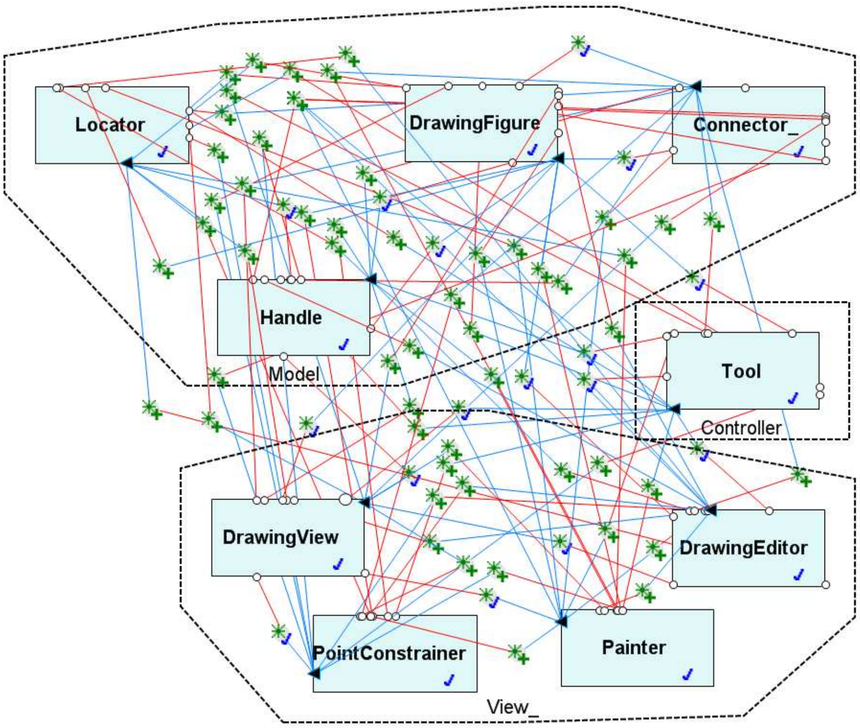


Figure 10: JHotDraw conformance results, with summary edges.

5.5 Extended Example: HillClimber

HillClimber is a 15,000 line Java application that was originally developed by undergraduates. HillClimber is interesting because it uses a framework, and its architectural structure had degraded over the years [2].

As-Designed Architecture. We based the as-designed HillClimber architecture on available documentation. In HillClimber, the application *window* uses a *canvas* to display *nodes* and *edges* of a *graph* to show the output of a computational *engine*. Based on a hint from one of the original framework developers, we posited that the `engine` component need not connect to `window` or `canvas`.

Annotation Process. The ownership annotations organized objects into a `data` domain to store the `graph`, a `ui` domain to hold user interface objects, and a `logic` domain to hold the `engine`, search objects, and associated objects. While adding annotations to HillClimber, we refactored the code to reduce coupling between `ui` and `data` objects [2]. The refactoring however did not affect the relationships between object in the `logic` and `ui` tiers.

Results. The conformance check confirms that `engine` connects to both `window` and `canvas`, contrary to the as-designed architecture (Fig. 14).

Metrics. The CCM is high since very few edges were affected (Table 3). The high node divergence is due to an as-designed view that has fewer elements at the top-level than the as-built view. The developer must either enrich the as-designed view by representing additional components in the `logic` tier, or refine the annotations to push more components in the `logic` tier into `engine`'s substructure.

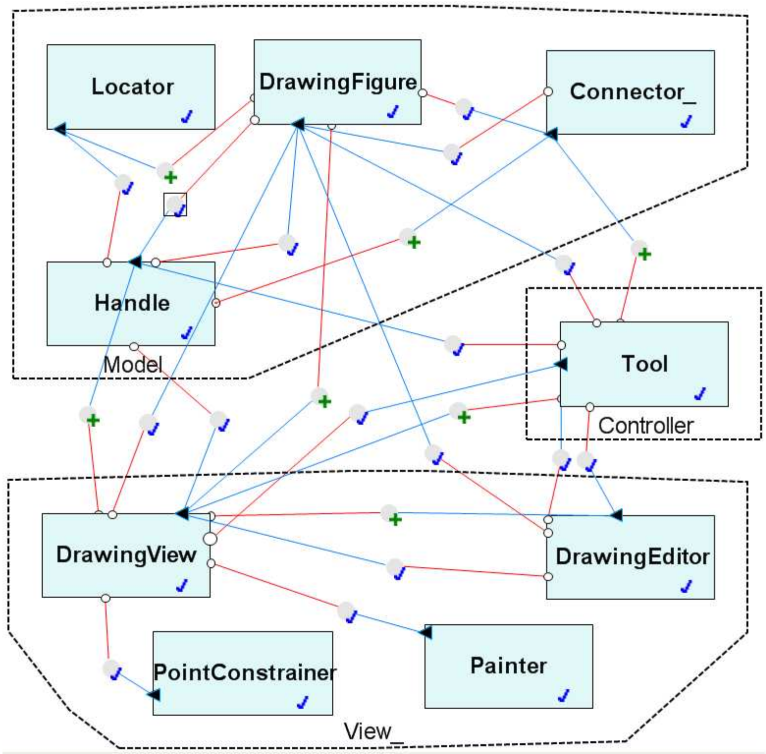


Figure 11: JHotDraw conformance results, without summary edges.

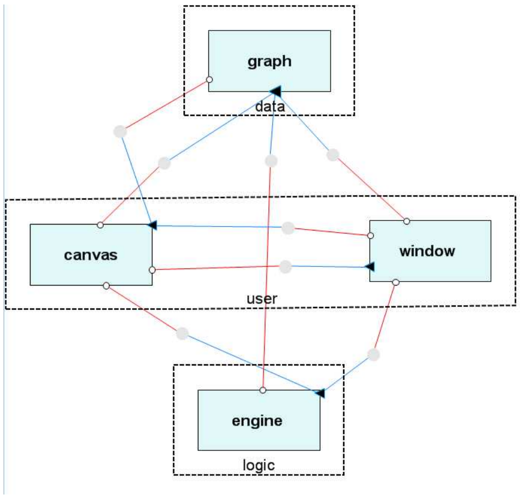


Figure 12: HillClimber as-designed view.

Table 3: HillClimber conformance metrics.

System	CN	DN	AN	CE	DE	AE	SE	CCM
HillClimber	4	14	0	10	2	0	12	83%

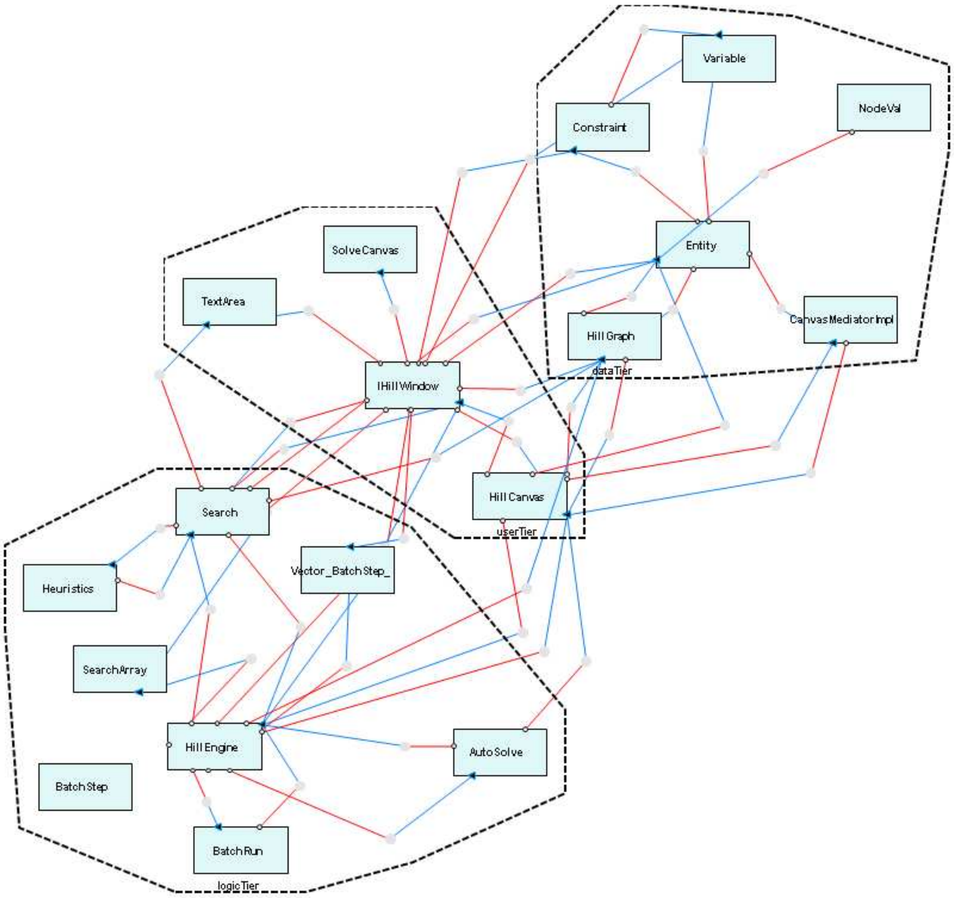


Figure 13: HillClimber as-built view.

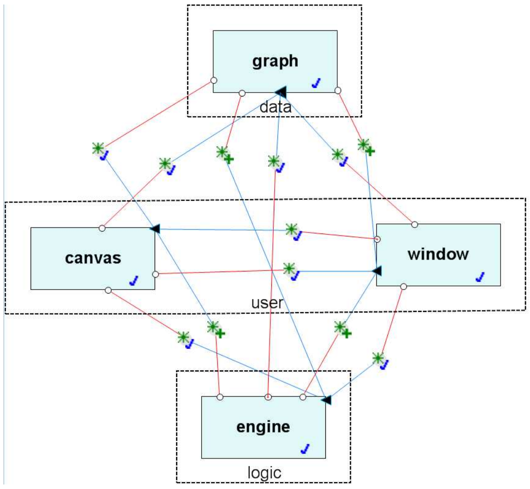


Figure 14: HillClimber conformance results.

6 Limitations

We discuss some limitations of the approach.

Annotations. Ownership inference is a separate problem and an active area of ongoing research [31, 26]. Previous ownership inference techniques can infer encapsulated objects in private domains and unaliased objects [25]. But they do not infer public domains, do not infer domain parameters [25] or infer too many parameters. So we used medium-sized programs under 50KLOC that we annotated manually.

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 bodies of code [33]. As a result, a study with an outside developer would be difficult given the nature of the approach. It may take a novice participant a least a week to learn the technique, add the annotations mostly manually, and apply the rest of the approach on a medium-sized system.

Architectural Extraction. The extracted C&C view is an approximation of the actual execution structure, one that is conservative and may show more communication than any system execution may have, by virtue of using a sound static analysis. Our evaluation indicates that the extracted architectures do not suffer from too much or too little abstraction [4]. However, our approach handles neither heterogeneous or distributed systems, nor dynamic architectural reconfiguration [34].

Structural Comparison. If the views are very different, the structural comparison may fail to match any components in the as-built view to the as-designed view. In that case, the comparison will not be useful since all components will be absences. One can then manually match the as-built and the as-designed 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.

7 Related Work

Hierarchical Views. Reflexion Models (RM) uses non-hierarchical high-level models and maps. Koschke et al. extended RM with hierarchical models [22]. The OOG is hierarchical and so is the as-built C&C view.

Edge Lifting. Many approaches that handle hierarchical models also account for substructure by lifting edges [22]. For example, in a code architecture, a summary edge is lifted from a function call to a module [39]. We use edge lifting in several places. By construction, an OOG lifts object relations from child objects to their parents. Converting an OOG to a C&C view also lifts some relations.

For example, an OOG can have an edge from `Displayer` to `Node` inside `Circuit`'s public domain `DB`. But the C&C view lifts that relation to circuit and shows a connector from Displayer to circuit.

Transitive Relations. Ommering et al. create a second module view that displays the transitive closure of a relation in one module view [39]. We compute the transitive closure of object relations, in the conformance view, and show them as summary connectors. Such a connector summarizes several connected objects with an edge. Similarly, when converting an OOG into a C&C view, we may add transitive edges to account for elided private domains.

Automated Mapping. Koschke et al. proposed semi-automated clustering algorithms to build an RM map [9]. In an evaluation on a code architecture, the engineer spent significant effort to derive a good partial mapping, and fine-tune the clustering parameters. An automated clustering may not always derive an architecture that is structurally comparable to a manually generated one. Our Aphyds evaluation showed how crucial that can be for a meaningful conformance check (21% vs. 57% CCM in Table 1).

Although better algorithms are needed, ownership annotations are amenable to type inference [25]. With precise and scalable inference tools, we believe our approach can scale to large systems.

Graph Comparison. Few conformance checking approaches compare the as-built and the as-designed view using a structural comparison. For instance, RM assumes that node names and optional token types match exactly. Unique node identifiers simplify the graph comparison considerably [11]. But without that assumption, our analysis can detect renames between the as-built and the as-designed views. This feature ensures that the conformance check is not sensitive to object labels. For example, an object in the OOG merges several field or variable declarations in the program. Each object reference that the program declares could have a different name or type. And the OOG nondeterministically selects a label for a given object o based on the name or the type of one of the object references that o represents.

Code Architecture. Several approaches capture various structural constraints on the code architecture. Sangal et al. enforce constraints on a module view using package dependency rules [35]. Feijs et al. check rules such as cycles [12] in a code architecture. Lague et al. compute metrics that compare the layers in the as-designed and the as-built code architectures [23]. Our approach is complementary, focuses on the runtime architecture, and computes metrics to relate the as-designed and the as-built runtime architectures.

Architectural Recovery. No previous technique recovers a hierarchical runtime view from a program in a general purpose language, entirely statically. Most approaches use a mix of dynamic and static analysis. As mentioned earlier, a dynamic analysis cannot prove that a program always satisfies a particular property. Our approach relies on machine-checkable ownership annotations, which enable it to soundly approximate the runtime instance structure, at compile-time.

In many approaches, the architectural abstraction is based on information such as directory structures, naming conventions, or various clustering algorithms [19], which take parameters that a developer can adjust. In our approach, a developer finely controls the abstraction with user-specified annotations. When we convert an OOG into an as-built C&C view, we exploit features such as the ownership hierarchy and the semantic distinction between private and public domains. In ARMIN, scripts can aggregate information, usually based on naming conventions, and produce higher-level views [20]. Currently, merging or splitting OOG objects into as-built C&C view components requires user interaction. More general user abstraction rules, similar to ARMIN, could be added in future work.

Language Solutions. Language-based solutions enforce conformance to a runtime architecture using a type system [8]. Similarly, the C2 ADL mandates a specific architectural framework [29]. But without following style guidelines, developers can still introduce architectural violations.

Conformance By Design. Generating an implementation from an architecture guarantees initial conformance [32]. Maintaining conformance requires always changing the specification then re-generating the implementation from the updated specification. Such approaches are often too restrictive and do not handle legacy code. Our post-hoc conformance checking handles legacy systems.

Object Graphs. Several analyses extract non-hierarchical object graphs statically without annotations [16]. Such an object graph does not convey architectural abstraction, and cannot be converted into an as-built view that is structurally comparable to an as-designed view.

Several dynamic analyses infer hierarchical object graphs without using annotations [15], but their results describe only the structure for those program runs. They also adopt a restrictive form of ownership. In ownership domains, an annotation can push almost any object underneath any other object in the ownership hierarchy. This expressiveness is crucial to avoiding an architecture with many objects in the top-level tiers. Our evaluation showed how crucial that can be for a meaningful conformance check.

Lam and Rinard proposed a type system and an analysis (which we call LR) that uses non-ownership annotations to extract a design from code [24]. LR extracts non-hierarchical object graphs that do not convey architectural abstraction. For Aphyds, LR would produce an object graph with even more top-level objects than Fig. 4(a).

Dynamic Analyses. DiscoTect [36] recovers from a running system an as-built C& view with architectural types. DiscoTect does not require annotations, but its results reflect only the particular inputs and exercised use cases. DiscoTect generates non-hierarchical C&C views that show one component for each instance created at runtime. Our OOG extraction and OOG-to-C&C conversion produce hierarchical as-built views that represent all possible program runs, as long as the annotations typecheck. DiscoTect does not check the conformance of the extracted views. The rest of our approach could apply to C&C views produced by DiscoTect.

Even though the cost of adding annotations is high, it is incurred once. In contrast, the cost of a dynamic analysis is incurred each time the analysis is run. The annotations can remain the program, and evolve with it. As the developers add new code or modify existing code, they can incrementally add annotations to the new code, and ensure that the annotations still typecheck.

View Synchronization. Our conformance analysis specializes our view synchronization work [6]. The key changes include processing the view differences asymmetrically (Section 4) and computing conformance metrics. For instance, even though the structural comparison detects renames, the conformance analysis rolls back the detected renames, to represent additional communication in the as-built view in terms of the as-designed view.

The hierarchical data used by the structural comparison now includes Groups, i.e., a Component or a Connector is a child of its owning Group. This extra level of hierarchy improves the precision of the comparison, and enables it to distinguish better between the Connectors within a given tier (which would belong to the same Group) and the ones that cross tiers (which would not be inside a Group).

8 Conclusion

We presented a semi-automated approach for statically checking the conformance between an as-designed and an as-built runtime architecture. The approach found interesting structural non-conformities in several real systems.

The approach uses annotations, supports existing object-oriented languages and designs, and does not require language extensions or implementation frameworks.

Acknowledgments. Larry Maccherone, William Scherlis and Mary Shaw gave us helpful advice on the approach. Nenad Medvidovic and Brad Myers gave us helpful comments on earlier drafts of this paper. Bradley Schmerl helped us with Acme and AcmeStudio.

References

[1] M. Abi-Antoun and J. Aldrich. Compile-Time Views of Execution Structure Based on Ownership. In *IWACO*, pages 81–92, 2007.

[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. Technical Report CMU-ISR-08-133, Carnegie Mellon University, 2008.

[4] M. Abi-Antoun and J. Aldrich. Static Extraction of Sound Hierarchical Runtime Object Graphs. Technical Report CMU-ISR-08-127, Carnegie Mellon University, 2008.

[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 and Software*, 80(2), 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*, 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. Storey. Equipping the Reflexion Method with Automated Clustering. In *WCRE*, 2005.

[10] P. Clements et al. *Documenting Software Architecture*. Addison-Wesley, 2003.
[11] 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.
[12] L. Feijs, R. Krikhaar, and R. van Ommering. A Relational Approach to Support Software Architecture Analysis. *Software - Practice & Experience*, 28(4), 1998.
[13] E. Gamma. Advanced Design with Patterns and Java (Tutorial). In *JAOO*, 1998.
[14] D. Garlan *et al.* The Acme Architectural Description Language. `http://www.cs.cmu.edu/~acme`.
[15] T. Hill, J. Noble, and J. Potter. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *Journal of Visual Languages and Computing*, 13(3):319–339, 2002.
[16] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *TSE*, 27(2), 2001.
[17] jRM. `http://jrmtool.sourceforge.net`, 2003.
[18] W. Kaiser. Become a programming Picasso with JHotDraw. JavaWorld, February 2001.
[19] R. Kazman and S. J. Carriére. Playing Detective: Reconstructing Software Architecture from Available Evidence. *Automated Software Eng.*, 6(2):107–138, 1999.
[20] R. Kazman, L. O'Brien, and C. Verhoef. Architecture Reconstruction Guidelines. CMU/SEI-2002-TR-034, 2002.
[21] J. Knodel and D. Popescu. A Comparison of Static Architecture Compliance Checking Approaches. In *WICSA*, 2007.
[22] R. Koschke and D. Simon. Hierarchical Reflexion Models. In *WCRE*, 2003.
[23] B. Lague, C. Leduc, M. Dagenais, A. L. Bon, and E. Merlo. An Analysis Framework for Understanding Layered Software Architectures. *IWPC*, 1998.
[24] 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.
[25] Y. Liu and A. Milanova. Ownership and Immutability Inference for UML-based Object Access Control. In *ICSE*, 2007.
[26] Y. Liu and S. Smith. Pedigree Types. In *IWACO*, 2008.
[27] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *TSE*, 21(9), 1995.
[28] J. McGarry et al. *Practical Software Measurement*. Addison-Wesley, 2001.
[29] 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*, pages 24–32, 1996.
[30] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *TSE*, 26(1), 2000.
[31] A. Milanova. Static Inference of Universe Types. In *IWACO*, 2008.
[32] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *TSE*, 21(4), 1995.
[33] 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.
[34] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. In *ICSE*, 1998.
[35] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using Dependency Models to Manage Complex Software Architecture. In *OOPSLA*, 2005.
[36] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering Architectures from Running Systems. *TSE*, 32(7):454–466, 2006.
[37] 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.
[38] M.-A. Storey, F. Fracchia, and H. Müller. Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration. *J. Syst.&Softw.*, 44(3), 1999.
[39] R. van Ommering, R. Krikhaar, and L. Feijs. Languages for formalizing, visualizing and verifying software architectures. *Computer Languages*, 27(1-3):3–18, 2001.