# Checking and Measuring the Architectural Structural Conformance of Object-Oriented Systems

**Marwan Abi-Antoun** **Jonathan Aldrich**

December 2007
CMU-ISRI-07-119

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Abstract**

The benefits of architectural analyses are only achieved if one can guarantee that the implementation conforms to the architecture. We propose an approach for checking and measuring the structural conformance of a software system's implementation to its execution architecture.

In contrast to existing approaches, our approach uses static analyses, and works with existing Java-like programming languages, existing object-oriented designs and existing integrated development environments.

We address the problem with a multi-pronged approach, as follows: (a) express and enforce architectural intent related to object encapsulation and communication directly in code using ownership domain annotations; (b) extract a sound execution architecture from the annotated program semi-automatically; and (c) compare the as-built extracted architecture to the as-designed architecture semi-automatically; and (d) obtain a measure of conformance.

We present an initial evaluation of the approach on two extended examples. In both cases, we extract as-built execution architectures that convey meaningful abstractions, convert them into standard component-and-connector architectures, and obtain measures of conformance between the as-designed and the as-built architectures that seem consistent with our intuition.

This work was supported in part by the United States Department of Defense.

**Keywords:** architectural extraction, architectural conformance, conformance measurement

# Contents

## List of Figures



## List of Tables

# 1 Introduction

Software architecture is concerned with capturing the structures of a system and the relationships among the elements both within and between structures [69]. A system can be described using several architectural views [69]. In particular, the *code architecture* shows the static source code organization. The *execution architecture* shows the system's organization into runtime components and their interactions. The execution architecture helps with reasoning about various runtime properties of a system, such as performance [71, 75], reliability [62], etc. But the benefits of architectural analyses are only achieved if the implementation conforms to the architecture, and the conformance problem has been identified as the "Achilles heel" of software architectures [35].

Many existing techniques have focused on enforcing the conformance to a code architecture [54, 42, 64], since extracting the execution architecture is a hard problem. Intuitively, it makes sense to use a dynamic analysis to recover the execution architecture, by instrumenting the running program, filtering the generated traces, then producing summary views [25, 67]. There are several problems with dynamic analyses. A dynamic analysis describes the execution architecture for one or more — but not all — program runs. As a result, changing the inputs, or executing different use cases, might produce different results. Thus, extracting the as-built architecture using a compile-time analysis would be ideal, if it can be achieved.

Recovering at compile-time the execution architecture of any system is hard. A static analysis for object-oriented programs must also deal with aliasing, recursion, inheritance, precision and scalability. In addition, to be most useful, the extracted execution architecture must be *sound*, i.e., it should not fail to reveal relationships that may exist at runtime.

Specifying a component-and-connector architecture directly in code, as in ArchJava [7], considerably simplifies the problem of enforcing the architecture in code, extracting the architecture from code, and ensuring conformance. Similarly, an implementation-constraining Architecture Description Language (ADL) with code generation capabilities [49], or an implementation-independent ADL with an implementation framework [45], such as C2SADL [48], may enable the compile-time extraction of the as-built execution architecture from an implementation. But such approaches impose non-backward compatible language extensions, or various restrictions on the implementation. Indeed, the effort to re-engineer existing Java implementations to ArchJava is non-negligible [7, 3].

Our proposed approach is more adoptable since it works with existing Java-like object-oriented programming languages, and existing object-oriented design idioms and patterns. The approach also works with code that uses existing frameworks and libraries, and does not require a specific implementation framework, e.g., [45]. The approach requires however adding annotations to the program. The idea of adding annotations to help recover a design from the code is not new, e.g., [43]. But existing annotations do not describe the runtime instance structure and data sharing precisely, or handle inheritance. Furthermore, these annotations are only descriptive, and do not enforce the architectural intent in code.

Any compile-time approach to extract an execution architecture must bridge the "dichotomy between the code structure (static hierarchies of classes) and the execution structure (dynamic networks of communicating objects)" [21]. So it seems plausible that richer type structures, e.g., ones that encode information about the runtime object structures into compile-time types, can help.

Many type systems have been proposed to enforce *ownership* at compile time [17, 55, 12, 6, 23, 60, 22, 53]. Making an object *owned* by, i.e., part of another object's representation, provides encapsulation guarantees and eliminates a source of bugs. In particular, an ownership type system enforces *instance encapsulation*, which is stronger than the module visibility mechanism of marking a class field as `private` — one can still define a `public` method that returns an alias to the object stored in the `private` field, thus causing a failure of encapsulation.

One of the challenges in extracting an execution architecture is the sharing of data between components. This sharing is often not explicit in object-oriented languages, but instead, is implicit in the structure of references created at runtime. The *ownership domains* type system [6] makes this sharing structure more explicit, by adding annotations to the reference types in the program. The annotations also constrain the communication through shared data, and ensure that the implementation conforms to those sharing patterns.

In previous work, we demonstrated how ownership domain annotations can express and enforce architectural intent in code in practice [6, 2]. We also demonstrated how the annotations enable the compile-time extraction of a sound execution architecture [1].

One can represent the execution structure of an object-oriented program as an *object graph*. Nodes correspond to objects, and edges correspond to relations between objects (See Figure 1(a)). Ownership domain annotations support abstract reasoning about data sharing by assigning each object at runtime to a single *ownership domain*, i.e., a *conceptual group of objects* (See Figure 1(b)). An object can declare in turn other domains, thus achieving hierarchy, and hierarchy helps with both high-level understanding and detail. In effect, each of these lower-level objects are folded into higher-level, architectural "component" instances (See Figure 1(c)).

A domain can correspond to an architectural *runtime tier*. A *tier* is a conceptual *partitioning* of functionality; sometimes, it identifies functionality that may be allocated to a separate physical machine, e.g., a `data` tier [10]. Most Architecture Description Languages (ADLs) have the notion of a tier or *group* [30, 20]. Existing object-oriented languages cannot specify this information directly in code. Similarly, most ADLs support the hierarchical decomposition of a component into a nested sub-architecture [49]. In ownership domains, each object can declare one or more domains to hold its internal objects, thus supporting hierarchical specification.

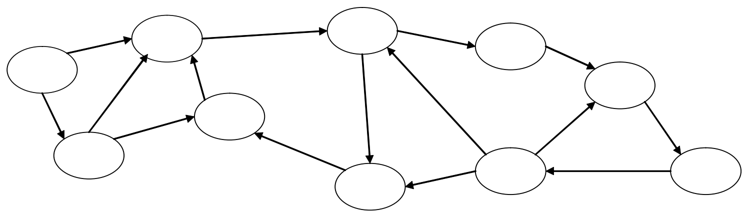
The annotations also describe policies that govern references between ownership domains. Objects within a single ownership domain can refer to one another. But references can only cross domain boundaries if there is a *domain link* between the two domains. Each object can declare a policy describing the permitted aliasing among objects in its internal domains, and between its internal domains and external domains. An ADL typically expresses such a policy using constraints [51].

In the proposed approach, we assume that a developer adds ownership domain annotations to the program under study, with or without tool support, for the benefit of other developers and architects. The annotations specify the architectural intent directly in code, enforce object encapsulation, and eliminate a common source of bugs due to the failure of encapsulation, as demonstrated by the existing research into ownership types.

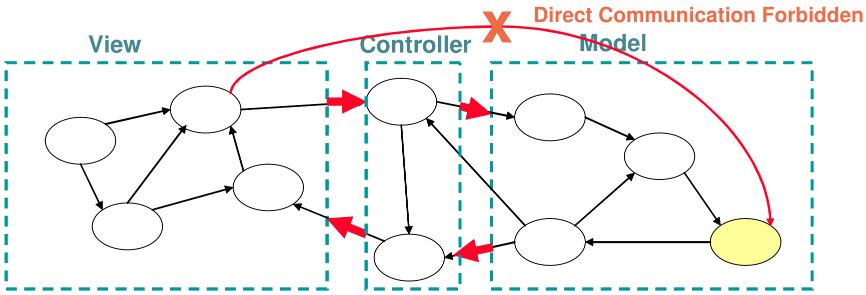
Existing static analyses that do not rely on annotations, tend to produce low-level, non-hierarchical object graphs [57, 36, 70]. Such object graphs explain runtime interactions in detail, but convey little architectural insight and do not scale. In contrast, our execution architecture is an instance-based, hierarchical runtime view of the system: it provides overviews of the system architecture at various levels of abstraction, by abstracting instances more effectively than a raw object graph. In addition, our execution architecture has hierarchy that corresponds to system decomposition in architectural diagrams [1].

One way to assure the as-built architecture is to extract it from an implementation, and compare it to the as-designed architecture [54, 4]. In this report, we assume that the as-designed architecture is a Component-and-Connector (C&C) view [18], represented in an Architecture Description Language (ADL). We then use our architectural differencing tool to detect renames, inserts, deletes, and restricted hierarchical moves between the two views [4]. Finally, a separate analysis uses the results of the structural matching to compute a measure of conformance between the as-built and the as-designed architectures.

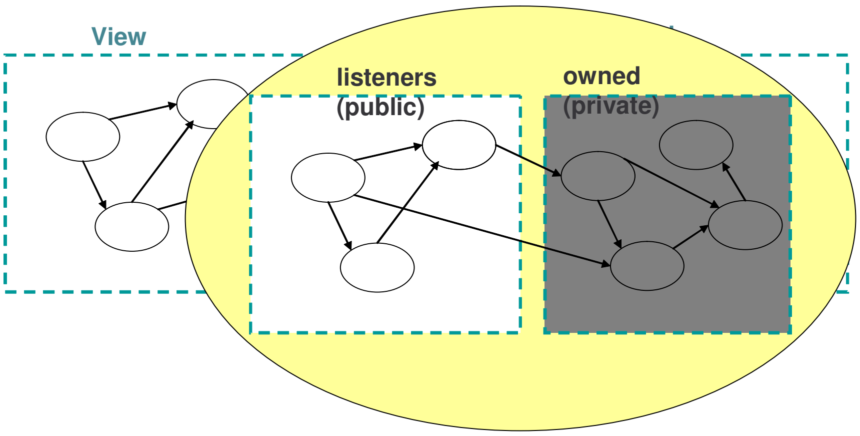
This report is organized as follows. In Section 2, we briefly discuss how ownership domain annotations express and enforce the architectural intent directly in code. In Section 3, we give the intuition behind the compile-time extraction of an execution architecture of a system from its annotated program. In Section 4, we discuss how we map the as-built architecture to the Acme ADL [30], to make it comparable to the as-designed architecture. In Section 5, we give the intuition behind our architectural differencing and merging tool. In Section 6, we use the tool's output to measure structural conformance. In Section 7, we evaluate the approach on a two real 15,000 line Java programs. We discuss some of the limitations of our approach in Section 8. Finally, we survey related work in Section 9 and conclude.



(a) An object-oriented program at runtime can be represented as a raw object graph. Nodes represent objects. Edges represent relations between objects.




(b) An *ownership domain* is a conceptual group of objects, and is shown as a dashed box. A *domain link*, shown as a thick arrow, abstracts communication permissions, i.e., objects communicate only when permitted. Permissions are not transitive, hence objects from the "View" domain cannot access objects in the "Model" domain in this contrived example.




(c) An object can declare in turn other domains, thus achieving hierarchy. In effect, each of these lower-level objects are folded into higher-level, architectural "component" instances.

Figure 1: Illustration of ownership domains and object graphs.

## 2 Specifying Architectural Intent in Code

We give the intuition behind the approach by example. CourSys is a simple course registration system to keep track of courses, students, and register students for courses. CourSys is implemented according to a three-tiered architectural style [68]. We will use CourSys as a running example throughout this report.

The first step in the proposed approach is to add ownership domain annotations to the program. Figure 2 shows the annotations that a developer might add to specify the architectural intent directly in code. While the concrete syntax [2] uses Java 1.5 annotations, we chose a simplified syntax to annotate the example (See Figure 2 for a summary).

The top-level class `Main` declares the ownership domains `user`, `logic`, and `data` (Line 36), corresponding to the tiers in a 3-tiered architecture style. The `Data` object instance is placed in the `data` domain, by annotating the corresponding field declaration, `objData` (Line 39). Similarly, the `Logic` object is placed in the `logic` domain, and the `Client` object is placed in the `user` domain (Lines 40, 41). Class `Main` also declares a domain link from domain `user` to domain `logic`, and a link from domain `logic` to domain `data` (Line 37). Due to the implicit permissions, a `Main` object can access the objects in the `client`, `logic` and `data` domains. But since permissions are not transitive, objects in the `user` domain cannot access objects in the `data` domain, for example.

Next, class `Logic` declares a default private domain named `owned` (Line 18). Class `Logic` also declares two objects, `log` and `lock`, in its `owned` domain (Lines 20,21). As a result, these objects are fully *encapsulated*, i.e., a programmer cannot write a public accessor method that returns any alias to one of these objects.

The `Data` object has references to state objects, such as `Student`, `Course`, etc. Obviously, these objects cannot be in a private domain of `Data`, since that would make them inaccessible to the outside. Instead, `Data` declares these objects in a *public domain*. As a result, any object that has access to a `Data` object has access to these state objects. For instance, `Data` declares the public domain `state` (Line 11) to hold the objects `Student`, `Course`, and lists thereof. Field `vStudent` (Line 12) is a reference to a list of `Student` objects, and is annotated with `state` — the outer `state` annotation is for the list object itself; the nested `Course<state>` annotation is for the actual list elements.

In fact, `Data` implements the `IData` interface, and most objects reference the `Data` object through its `IData` interface. As a result, the interface `IData` also declares the public domain `state`. The latter gets unified with the one that `Data` declares.

The `Logic` object needs references to the state objects, such as `Student`, `Course`, etc., that are in another domain. So `Logic` declares a *domain parameter*, to receive access to another object's state (Line 17). An instance of `Logic` gets access to the state of an instance of `Data`, when `Main` binds `Data`'s public domain `state`, to `Logic`'s domain parameter, `state`. An object's public domain is treated a field, and accessed using a similar syntax, and the binding of a formal domain to a domain parameter uses a syntax similar to Java generics (Line 40). As a result, `Logic` accesses objects `Student`, `Course`, etc., in `Data`'s public domain, `obj.Data`. In addition, a `Logic` object maintains a persistent reference to the `Data` object itself to retrieve the state data from a repository. So `Logic` declares another domain parameter, `dataTier`, to get a reference to the `Data` object. The `dataTier` formal domain parameter on `Logic` is bound to the actual `data` domain of object `Main` (Line 40).

When programming to an interface instead of a concrete implementation class, the code references the `Logic` object through its `ILogic` interface. So, the `ILogic` interface is also parameterized by an `istate` domain parameter (Line 6). Class `Logic` binds the `istate` domain parameter it inherits from `ILogic` (Line 6), using a syntax similar to generics (Line 17).

**Summary: The architectural intent is specified directly in code using ownership domain annotations. The extracted architecture reflects the annotations, and the quality of the extracted architecture reflects the quality of the annotations. The architectural intent is expressed by choosing the ownership domains and their structure, then adding annotations to the program. The annotations must be added in such a manner as to make the extracted architecture similar to the as-designed one. The annotations are currently added manually, though active work in annotation inference promises to lower the annotation burden.**

```
interface IData {
  domain state; /* Public domain */
  state ArrayList<Course<state>> getAllCourseRecords();
  ...
}
interface ILogic <istate> {
  shared String getAllCourses();
  ...
}
class Data <logicTier> implements IData {
  domain state; /* Public domain -- gets unified with one on IData interface */
  state ArrayList<Student<state>> vStudent;
  state ArrayList<Course<state>> vCourse;
  ...
  logicTier ILogic<state> logic; /* Create a reference back */
}
class Logic <dataTier,state> implements ILogic <state> {
  private domain owned; /* Default */
  dataTier IData dataNode;
  owned Logging log;
  owned RWLock lock;
  ...
}
class Client <logicTier,state> {
  logicTier ILogic<state> logicNode;
  ...
}
class Main {
  domain user, logic, data;
  link user->logic, logic->data;

  data final Data<logic> objData; /* Must be final to access public domain */
  logic Logic<data,objData.state> objLogic; /* Use public domain */
  user Client<logic,objData.state> objClient; /* Use public domain */

  // Create reference between objData and objLogic
  objData.logic = objLogic;

  ...
}
```

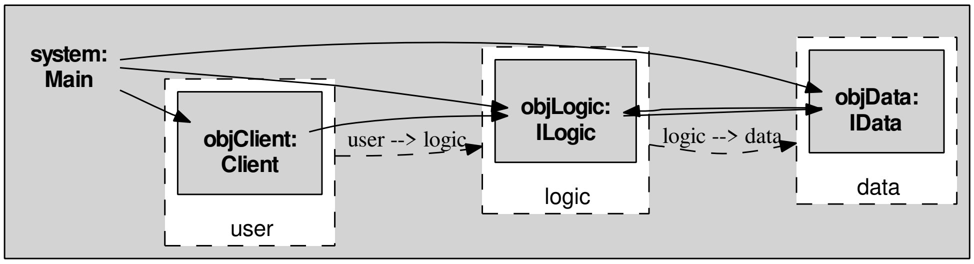
(a) Java code snippets from the CourSys example with ownership domain annotations.

`d T o`: declare object `o` of type `T` in domain `d`;
[public] **domain** `a`: declare private [or public] domain;
`class C<d>`: declare formal domain parameter `d` on class `C`;
`C<actual> cObj`: provide actual for domain parameter;
**link** `b -> d`: give domain `b` access to domain `d`;
**Special Alias Types.** A few special annotations add expressiveness to the type system:

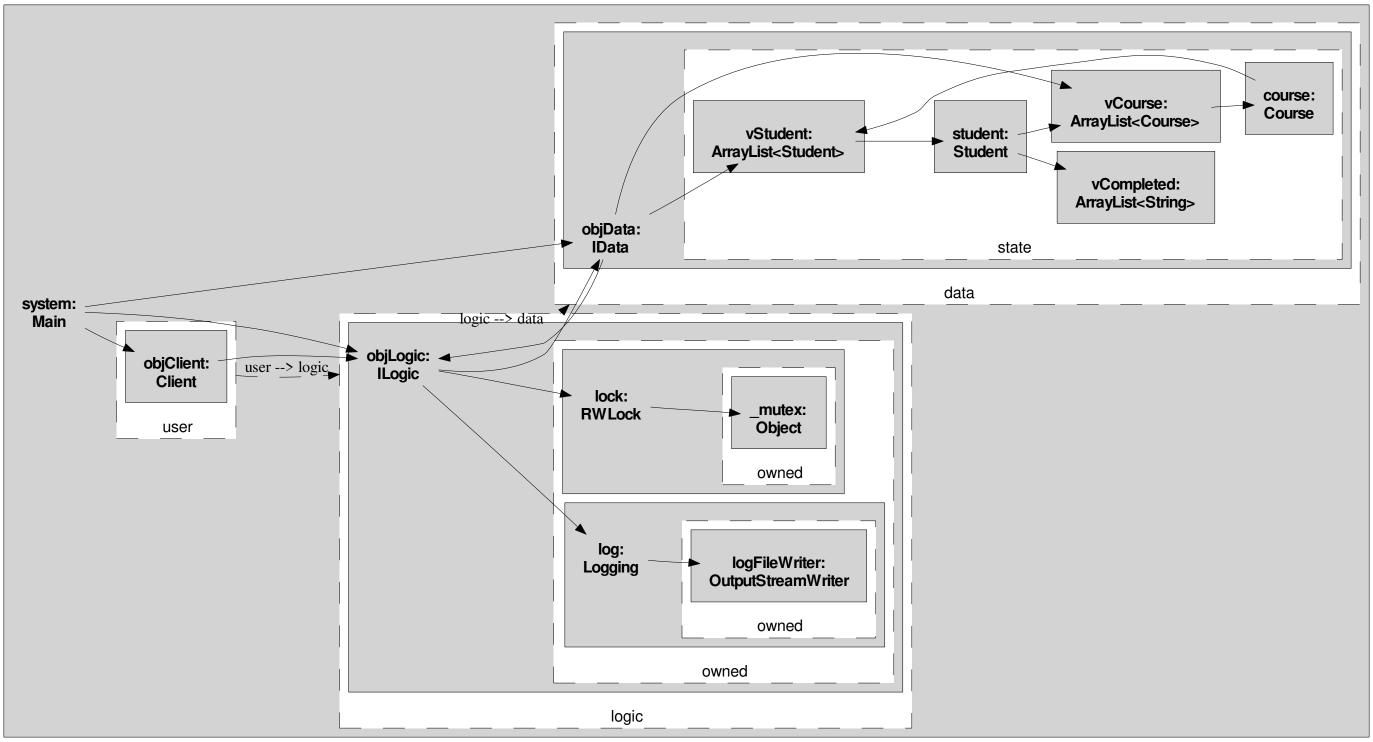
- `unique`: indicates an object to which there is only one reference, such as newly created objects. Unique objects can be passed linearly from one object to another;
- `lent`: one ownership domain can temporarily lend an object to another ownership domain, and ensure that the second ownership domain will not create any persistent references to the object;
- `shared`: the object may be aliased globally. `shared` references may not alias non-`shared` references.

(b) Simplified syntax for ownership domain annotations. The concrete syntax and other details are available elsewhere [2].

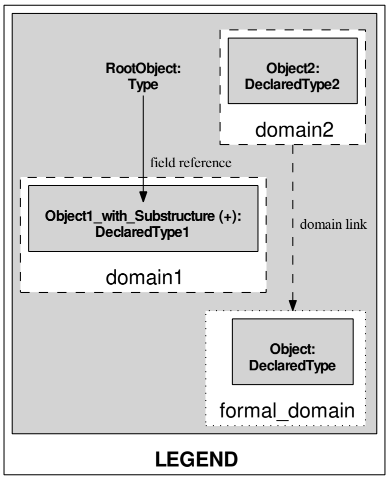
Figure 2: CourSys source code with ownership domain annotations.




(a) Top-level execution architecture for CourSys, laid out automatically.




(b) Second-level execution architecture for CourSys, showing `objLogic`'s substructure, and that of `log` and `lock`.




(c) Ownership Object Graph legend.

Figure 3: CourSys execution architecture at two levels of abstraction.

# 3 Extracting the As-Built Execution Architecture

## 3.1 Illustrative Example

In our approach, a static analysis extracts from the annotated program in Figure 2 the execution architecture shown in Figure 3(a). Figure 3(a) follows the graphical conventions in Figure 3(c). A dashed border, white-filled, rectangle represents an actual ownership domain. A solid border, grey-filled, rectangle with a bold label represents an architecturally-relevant object or component instance (which can have substructure). The substructure can be shown either using nested domains and objects inside them. Or it can be elided, in which case, the (+) symbol is appended to the object's label. A dashed edge represents a domain link between two ownership domains. A solid edge can represent a creation, usage, or reference relation between two objects. An object is labeled as `obj : T`, where `obj` is the name of the field or variable declaration, and `T` is a declared type, as in UML object diagrams.

Figure 3(a) shows explicitly the tiered execution architecture. Object `objClient` in a `user` tier. Object `objLogic` is in a `logic` tier. Object `objData`, objects of type `Student`, `Course` and lists thereof, are in a `data` tier. There are domain links from domain `user` to domain `logic`, and from domain `logic` to domain `data`. The edges in Figure 2 correspond to field references.

In Figure 3(a), each gray box corresponds to a "canonical object" that represents many instances at runtime, and has instance substructure. This corresponds closely to the system decomposition typically seen in an architectural diagram. The execution architecture in Figure 3(a) folds lower-level objects into higher-level, architectural component instances. As a result, the execution architecture provides abstraction, by not showing non-architectural instances in the top-level domains. Collapsing many nodes into one is a classic approach to shrink a graph. However, our execution architecture is unique in collapsing nodes based on actual execution and ownership structures, and not according to where objects are declared are in the source code.

Figure 3(a) shows object ownership hierarchy, which is different from the hierarchy found in class diagrams. Class diagrams often use packages to organize related classes. For instance, placing the `Logic` and the `Logging` classes in a `coursys.logic` package indicates that they belong to the same layer in the code architecture[1]. The execution architecture indicates that an instance of `Logging` is *owned* by an instance of `Logic`, i.e., hidden in its representation and inaccessible to other objects at runtime — even to other objects in the `logic` tier.

Figure 3(a) shows only the top-level domains and the objects inside them. Figure 3(b) shows additional substructure, by increasing the depth of the object hierarchy being viewed. In particular, Figure 3(b), shows objects `log` and `lock` inside `objLogic`. Furthermore, it shows objects `mutex` and `logFileWriter` inside `lock` and `log`. The flexibility of the type system allows the grouping of objects according to logical containment, and not complete encapsulation. For instance, the `logic` tier accesses `Course` and `Student` objects in the `data` tier.

There are other ways to annotate CourSys. For this approach to work, the annotations must be added in such a manner as to make the extracted architecture similar to the as-designed one. For instance, we did not have to place the state objects in a public domain of the `Data` object, but could have placed them instead in the `dataTier` (See Figure 4).

## 3.2 Annotation-Based Architectural Recovery

Storey and Müller argue that "the process of building mental hierarchical abstractions from the low-level software objects and relations is the hardest part of bottom-up comprehension for many maintainers, and yet many tools only support showing a previously abstracted view". They add that "maintainers might understand the software better through abstractions they created themselves, rather than through the pre-fabricated abstractions that many tools provide. Facilities should be available to allow the maintainer to create their own abstractions and label and document them to reflect their meaning" [72].

[1]Following Clements et al., we adopt a precise terminology as follows: a *layer* denotes a cluster or a partition in the *code architecture* or a *module view*. A *tier* denotes a cluster or a partition in the *execution architecture* or a *runtime view* [18].

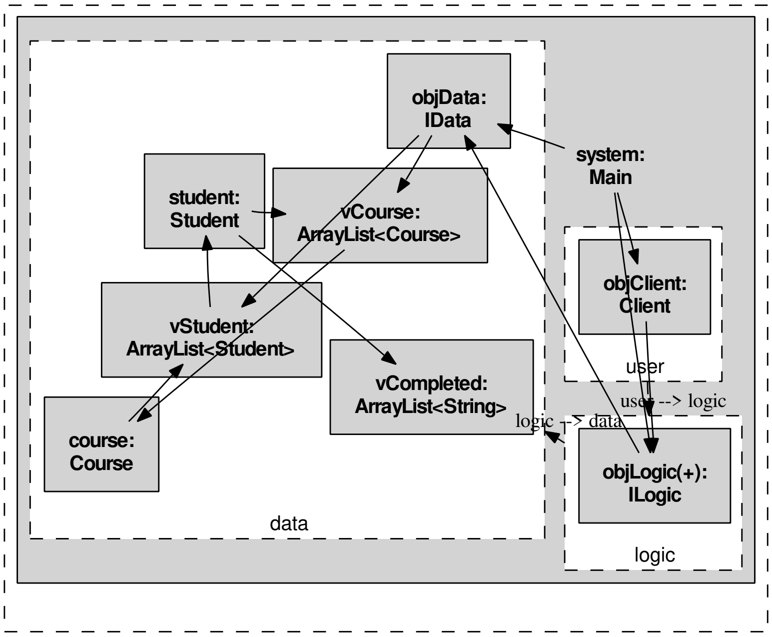



Figure 4: CourSys alternate architecture extracted from a different set of possible annotations.

In our approach, the developer controls the abstraction through source code annotations[2]. In most cases, different abstractions could be obtained by changing the annotations without necessarily changing the source code.

The ownership domains type system supports pushing any object underneath any other object in the ownership hierarchy. In ownership domains, 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. This is in contrast to an *owner-as-dominator* type system e.g., [17], which requires any access to a child object to go through its owning object. This expressiveness is crucial for avoiding an execution architecture that has too many top-level objects. If making an object owned by another object restricts access to the owned object, this forces more objects to be peers.

**Summary: Ownership domain annotations enable extracting, at compile-time, from an annotated program, an execution architecture of the system. The extracted execution architecture is sound, i.e., it does not fail to reveal relationships that actually exist at runtime.**

[2]The current annotation-based system adds Java 1.5 annotations to the program, and uses external XML files to annotate external libraries [2]. All the annotations could, at least in principle, be in external XML files.

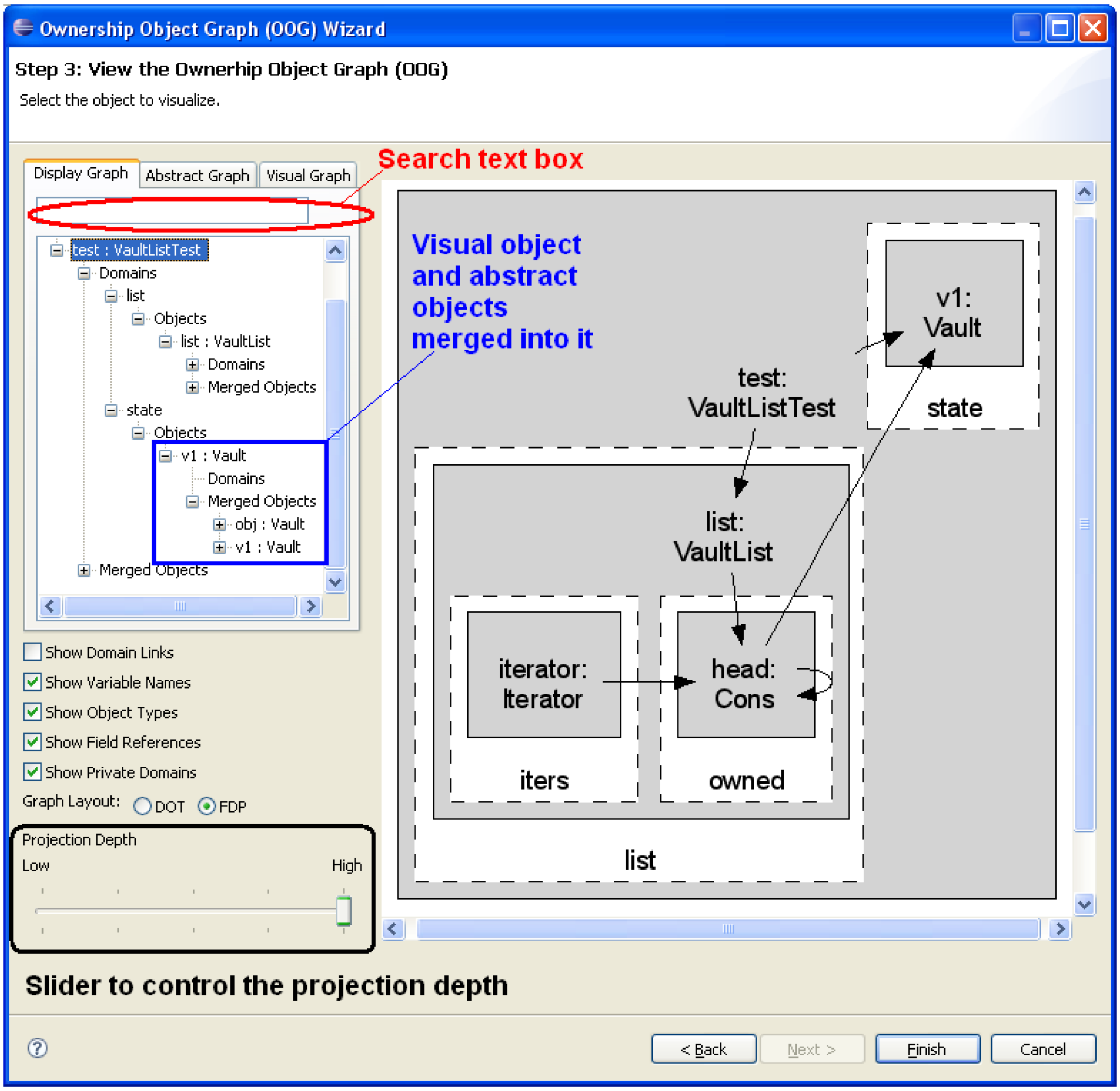



Figure 5: Screenshot of the architectural extraction tool.

# 4 Mapping to an Architecture Description Language

We assume that the as-designed architecture is a Component-and-Connector (C&C) view [18], documented in an Architecture Description Language (ADL). We also map the as-built architecture to a C&C view, to make it comparable to the as-designed architecture.

## 4.1 Review of the Acme

We use a standard general purpose Architecture Description Language (ADL), Acme [30]. We briefly review the Acme design elements:

- Component: a Component represents a unit of computation and state in the system. Intuitively, a Component corresponds to a box in boxes-and-lines description of a software architecture;
- Port: a Port represents a point of contact between the component and its environment. A Port often represents what is traditionally thought of as an interface, i.e., a set of operations available on a component. A Component may have several ports corresponding to different interfaces to the component;
- Connector: a Connector represents an interaction among components;
- Role: a Connector includes a set of interfaces in the form of Roles. Each Role defines a participant in the interaction captured by the Connector;
- Attachment: an Attachment represents a connection between a Port and a Role;
- System: a System represents a configurations of Components and Connectors;
- Representation: Acme supports the hierarchical description of architectures. Specifically, any component or connector can be represented by one or more detailed, lower-level descriptions. Each such description is termed a Representation in Acme.
- Binding: a Binding ties the inside of a Representation to its outside;
- Property: any of the Acme architectural design entities can be annotated with a set of properties. In Acme, each Property is a name and value pair;
- Group: a Group is a conceptual group of Acme design elements. It can also have properties.

## 4.2 Mapping to Acme

We next discuss how the elements from the extracted architecture are mapped to Acme design elements.

**System.** The root object is mapped to the corresponding architectural System. The System is assigned the TieredFam architectural style, one of the Acme predefined styles, that is used to represent a tiered architecture.

**Components.** Each object in the OOG is mapped to an architectural Component. Each component is assigned the TierNodeT type from the TieredFam style.

**Component Ports.** Object relations in the OOG are mapped to ports as follows. If `objA` has a field reference of type `T` to Object `objB`, the Component corresponding to `objA` has a Port of type useT, and name `objB`. The Component corresponding to `objB` has a Port that provides services of type provideT, and name `T`. By convention, we only create unidirectional ports, i.e., a port has either type provideT or useT, and not both.

**Connectors.** A relation between objects maps to an architectural Connector. We do not map self-edges in the OOG since they do not seem architecturally interesting. Each Connector is assigned the CallReturnT type from the TieredFam style.

**Connector Roles.** Connector roles are created to be compatible with the ports to which they are attached.

- **Case #1:** A Role attached to a Port of type provideT has type providerT;
- **Case #2:** A Role attached to a Port of type useT has type userT.

**Groups.** An ownership domain in the OOG is mapped to a Group. If an object $o$ in a domain $d$, the corresponding Component is in Group $g$.

Both domains and groups are "conceptual groups" of objects or components. There are however several differences between a domain and a group. A component does not have to be in a group or can be in multiple groups. On the other hand, every object is always in exactly one domain. The constraint that a component is at most in one tier can be enforced using a rule on the corresponding Group, as follows[3]:

```
forall g1 in self.groups |
    forall g2 in self.groups |
        forall m1 in g1.members |
            forall m2 in g2.members |
            m1 = m2 -> g1 = g2;
```

**Hierarchy.** An object in the OOG has hierarchical sub-structure, which maps to system decomposition. More specifically, i an object declares domains, the corresponding Component has a Representation. and a sub-architecture inside that Representation.

**Connectors in Groups.** If a Connector *conn* attaches two components $comp_1$ and $comp_2$, and both $comp_1$ and $comp_2$ are in a Group $g$, then *conn* is automatically added to $g$. This reduces the number of connectors at the top-level, and improves the match precision.

**Domain Links.** Domain links are policies between ownership domains. Acme does not have the notion of a first-class edge between two Groups. However, a Group can have properties and rules associated with it. For example, to specify a link between two Groups, one possibility is to use a property, which defines the list of groups that the current group is linked to:

```
    group logicTier = {
        Property linkedTo : string = "dataTier";
    }

    group dataTier = {
    }
```

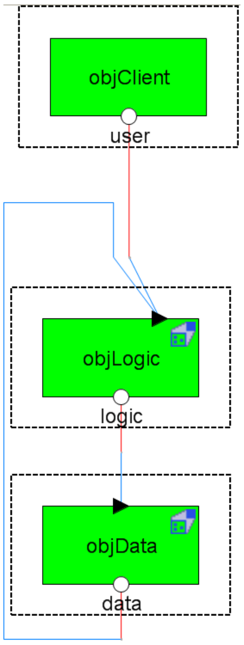
**Reserved Keywords.** Acme has many reserved keywords, such as "Component", "Connector", "View", "In", etc. When mapping to Acme, we automatically rename elements to avoid a name clash with a reserved keyword: e.g., "View" is renamed to "View_".

## 4.3 Illustrative Example

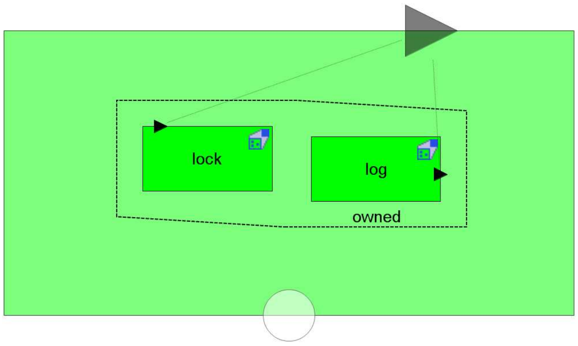
We illustrate the mapping of the CourSys OOG to an Acme architecture in Figures 6(a),6(b),6(d).

**Summary: A tool can map the extracted architecture to a Component-and-Connector (C&C) view, represented in a standard Architecture Description Language (ADL), such as Acme. The domains in the extracted architecture map intuitively to Groups in the ADL. Architecturally relevant objects map to Components. Relations between objects map to Connectors, Ports and Roles. The hierarchy maps naturally to system decomposition.**

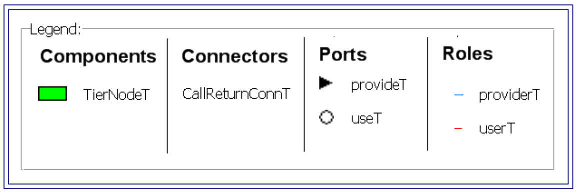
[3]The current implementation does not yet support `self.groups`, but should be fixed soon.



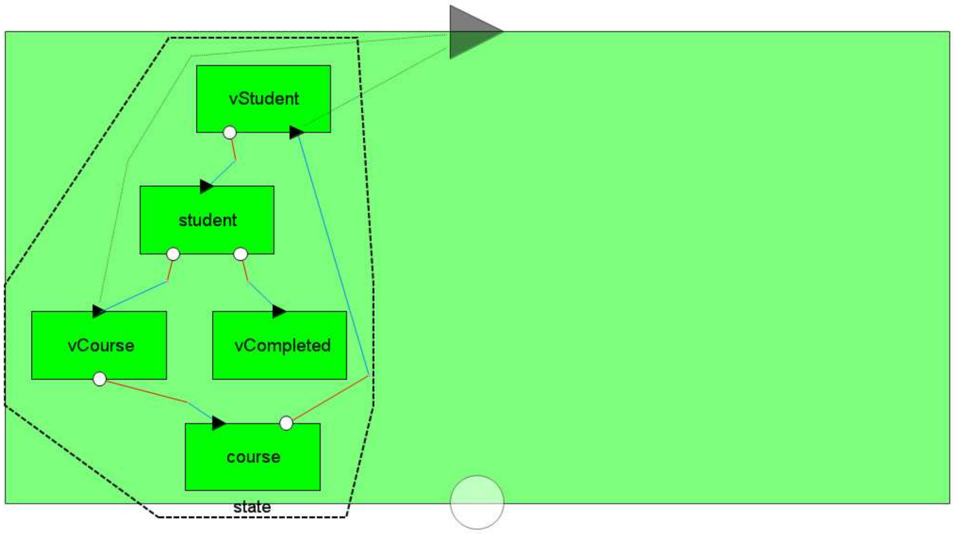

(a) Top-level system.




(b) Substructure of component `objLogic`.




(c) Legend for the Acme `TieredFam` style.




(d) Substructure of component `objData`.

Figure 6: CourSys extracted architecture represented in the Acme ADL.

# 5 Checking Conformance

We compare the as-built architecture to the as-designed one semi-automatically using our existing tool for differencing architectural views [4]. We briefly summarize the tool's capabilities in this section.

## 5.1 Problem Definition

A architectural view is generally described as a graph, so view differencing and merging is a problem in graph matching. Graph matching measures the similarity between two graphs using the notion of graph edit distance, i.e., it produces a set of edit operations that model inconsistencies by transforming one graph into another [19]. Typical graph edit operations include the deletion, insertion and substitution of nodes and edges. Each edit operation is assigned a cost. The costs are application-dependent, and model the likelihood of the corresponding inconsistencies. Typically, the more likely a certain inconsistency is, the lower is its cost. Then the edit distance of two graphs $g_1$ and $g_2$ is found by searching for the sequence of edit operations with the minimum cost that transform $g_1$ into $g_2$. A similar problem formulation can be used for trees. However, tree edit distance differs from graph edit distance, in that operations are carried out only on nodes and never directly on edges.

Graph matching is NP-complete in the general case. Unique node labels enable processing graphs efficiently which explains why many approaches make this assumption. Optimal graph matching algorithms, i.e., those that can find a global minimum of the matching cost if it exists, can handle at most a few dozen nodes. Non-optimal heuristic-based algorithms are more scalable, but often place other restrictive assumptions.

**Challenges and Requirements.** Several efficient algorithms have been proposed for trees, a strict hierarchical structure. Our tool leverages the fact that many architectural views hierarchical. While not all architectural views are hierarchical, many use hierarchy to attain both high-level understanding and detail. In a C&C view, the tree-like hierarchy corresponds to the system decomposition, but cross-links between the system elements form a general graph. Other architectural views, such as module views, have similar characteristics [18]. We relax however the constraints of previous approaches. For a more detailed discussion, refer to earlier publications [4, 5]:

- **No Unique Identifiers.** For maximum generality, we do not require elements to have unique identifiers, as in other approaches;
- **No Ordering.** In the general case, an architectural view has no inherent ordering amongst its elements. This suggests that an unordered tree-to-tree correction algorithm might perform better than one for ordered trees. Many efficient algorithms are available for ordered labeled trees. Some algorithms for unordered trees achieve polynomial-time complexity either through heuristic methods or under additional assumptions.
- **Renames.** A synchronization approach must of course handle elements that are inserted and deleted, as supported by ArchDiff [14]. But effective synchronization must also go beyond insertions and deletions, and support renames. Identifying an element as being deleted and then inserted when, in fact, it was renamed, would result in losing crucial style and property information about the element, even if this produces structurally equivalent views. These architectural properties, such as throughput, latency, etc., are crucial for many architectural analyses, e.g., [71]. In the following discussion, a *matched* node is a node with either an *exactly matching* label or a *renamed* label.
- **Hierarchical Moves.** Architects often use hierarchy to manage complexity. In general, two architects may differ in their use of hierarchy: a component expressed at the top level in one view could be nested within another component in some other view. This suggests that an algorithm should detect sequences of *internal node deletions* in the middle of the tree, which result in nodes moving up a number of levels in the hierarchy. An algorithm should also detect sequences of *internal node insertions* in the middle of the tree, which result in nodes moving down in the hierarchy, by becoming children of the inserted nodes.
- **Manual Overrides.** Structural similarities may lead a fully automated algorithm to incorrectly match top-level elements between two trees and produce an unusable output. Because of the dependencies

in the mapping, one cannot easily correct these incorrect matches after the fact. Instead, we added a feature not typically found in tree-to-tree correction algorithms. The feature allows the user to force or prevent matches between selected view elements. The algorithm then takes these constraints into account to improve the overall match. The user can specify any set of constraints, as long as they preserve the ancestry relation between the forcibly matched nodes. In particular, if $a$ is an ancestor of $b$, $a$ is forcibly matched to $c$, and $b$ is forcibly matched to $d$, then $c$ must be an ancestor of $d$.

- **Optional Type Information.** Architectural views may be untyped or have different or incompatible type systems. This is often the case when comparing views at different levels of abstraction, such as an as-designed conceptual-level view with an as-built implementation-level view. Therefore, an algorithm should not rely on matching type information, and should be able to recover a correct mapping from structure alone if necessary, or from structure and type information if type information is available. An algorithm could however take advantage of type information, when available, to prune the search space by not attempting to match elements of incompatible types.
  If the view elements are represented as typed nodes, at the very least, an algorithm should not match nodes of incompatible types, e.g., it should not match a connector $x$ to a component $y$. If architectural style information is available, additional architectural types may be available and could be used for similar purposes. For instance, an algorithm can avoid matching a component of type `Filter`, from a Pipe-and-Filter architectural style, to a component of type `Repository`, from a Shared-Data architectural style [68].
- **Post-hoc Comparison.** For maximum generality, we assume a disconnected and stateless operation. A few approaches require monitoring or recording the structural changes while the user is modifying a given view.

## 5.2 Comparing Architectural Views

So far, we have extracted the as-built architecture from code (Section 3), and mapped it to an Architecture Description Language (ADL) (Section 4). We also represented the as-designed architecture in an ADL. We now use our architectural differencing and merging tool to structurally compare the as-built architecture to the as-designed one.

We represent the structural information in a C&C view as a cross-linked tree structure that mirrors the hierarchical system decomposition. The tree also includes some redundant information to improve the accuracy of the structural comparison. For instance, the subtree of a node corresponding to a port includes additional nodes for all the port's involvements, i.e., all the components and their ports reachable from that port. Each node is decorated with properties, such as type information. The type information, if provided, populates a matrix of incompatible nodes that may not be matched. That matrix also includes optional user-specified constraints to force or prevent matches.

A graph representing a C&C view can generally have cycles in it. Representing an architectural graph as a tree causes each shared node in the graph to appear in several subtrees. We consider one of these nodes as the *defining occurrence*, and add a *cross-link* from each repeated node back to its defining occurrence. These redundant nodes, while they significantly increase the size of the corresponding trees under comparison, greatly improve the accuracy of the tree-to-tree correction. However, they may be inconsistently matched with respect to their defining occurrences, either in what they refer to, or in the associated edit operations.

We work around these inconsistent matches using two passes. During the *first pass*, we synchronize the strictly hierarchical information corresponding to the system decomposition, i.e., components, ports and representations. During the *second pass*, we synchronize the edges in the architectural graph. The post-processing step is simple at that point, since it knows the mapping between the nodes in the two graphs.

**Assumptions.** We make the following assumptions:

- **The views are comparable.** The two views under comparison have to be somewhat structurally similar. When comparing two completely different views, an algorithm could trivially delete all elements of one view, and then insert them in the other view. In addition, the two views must be of the same viewtype, and must be comparable without any view transformation;

- **There are no merged or split elements**. Our approach does not currently detect the merging or splitting of view elements. Merging and splitting are common practice, but are difficult to formalize. We leave merges and splits to future work;
- **The hierarchical information is correctly matched first.** We assume that the strictly hierarchical information corresponding to the system decomposition, i.e., components, ports and representations, is first correctly matched. If that cannot be achieved through structural comparison alone, the user has to manually force the matches between the top-level elements, and re-run the synchronization. This ability to force and prevent features is one of the novel features of our algorithm [4];
- **Type information, when available, is used to prevent the matching of elements of incompatible types.** This helps with correctly matching ports and roles of the right types, as follows
  - A Port of type useT cannot be matched to a Port of type `provideT`, and vice versa;
  - A Role of type userT cannot be matched to a Role of type `providerT`, and vice versa;

**Common Super-Tree.** From the edit script, the synchronization produces a common super-tree that previews the merged view after the edit actions are applied. In particular, each node in the supertree has a status code, e.g., RENAME, INSERT, DELETE, and a reference to the matched model element in the other view (this would be not applicable for elements with an INSERT status).

**Recent Changes.** We made the following changes to our earlier synchronization tool [4]:

- **Add groups to the tree structured data:** we added groups to the tree-structured data. Components and connectors are added as children to the owning group, if they belong to one. If not, they appear as top-level components and connectors. Having the additional group hierarchy reduces the number of elements at the top-level and improves the match precision. However, the approach still works if the as-designed architecture does not use groups.
- **New actions to add/remove element to/from group:** since groups are now part of the tree structured data, some edit actions can add or remove a component or a connector from a group.
- **New conformance visitor:** we implemented a conformance visitor separate from the synchronization visitor, that traverses the common supertree, computes the conformance output and the conformance metrics (the latter are discussed in Section 6).

Checking conformance is slightly different from view differencing and merging. In the full synchronization scenario, all the changes in the as-built architecture are pushed to the as-designed architecture. This approach is not satisfactory for checking conformance, for the following reasons.

- **Additional sub-structure in the as-built architecture:** the as-built, implementation architecture is likely to have more details compared to the as-designed view. An as-designed architecture may omit unnecessary details, e.g., the detailed substructure of a component.
- **Inconsequential renames in the as-built architecture:** the as-built implementation architecture is likely to have many renames compared to the as-designed one. But typically, the names in use in the as-designed architecture map better to the architect's conceptual mental model. The names in the as-built architecture may be restricted by various implementation idiosyncrasies, e.g., to avoid a name clash with a framework or library in use.

## 5.3 Conformance Strategies

There are different design choices on determining the conformance between the as-built and the as-designed architecture. The main question has to do with how to treat additional details in the as-built architecture.

- **Strategy #1: Only show the connections that are missing from as-built architecture.** In this strategy, we assume that we only care about the components in the as-designed architecture, and want to ignore any additional top-level components in the as-built architecture.

  The main advantage of this strategy is that it allows the as-designed architecture to truly represent the architect's view, and possibly elide information that is outside of a specific concern, e.g., security.

This strategy is not entirely satisfactory. What if the following scenario exists? As-designed ComponentA communicates with missing ComponentC — ComponentC is in the as-built but not in the as-designed — and ComponentC talks to the as-designed ComponentB. If we do not add the missing ComponentC to the as-designed architecture, we must still add a "summary edge" showing the covert communication between the as-designed components, ComponentA and ComponentB. Such a scenario is important from a conformance or a security standpoint.
In another scenario, the as-designed architecture may not include a missing ComponentD and a missing ComponentE, and ComponentD and ComponentE only communicate with each other. Not showing that communication seems less relevant from a conformance standpoint.

- **Strategy #2: Show all the top-level components and connections from the as-built architecture.** In this strategy, we assume that the as-designed architecture must be a faithful and complete description [4]. However, if an as-designed ComponentA does not have substructure, then we ignore any substructure in the corresponding as-built component. But if as-designed ComponentA has substructure, then we check the substructure of its as-built counterpart.
  This strategy is more principled than Strategy #1 since it does not arbitrarily elide components and connections. Its main disadvantage however is that it may lead to as-designed architectures that are too detailed. Moreover, the strategy might require the developer to add very precise annotations, to reduce the number of top-level objects in the extracted execution architecture. In some cases, the effort required to reduce the clutter may not be justified.
- **Strategy #3: Support user abstraction rules when converting the extracted execution architecture to a C&C view.** In this strategy, the user controls how the extracted architecture is mapped to a C&C view, that is used for checking conformance against the as-built view. For instance, the user can map an entire ownership domain to a Component. Similarly, the user can merge two components in the as-built view into one component in the as-designed view.

Finally, Strategies #2 and #3 can be combined.

## 5.4 Computing Conformance

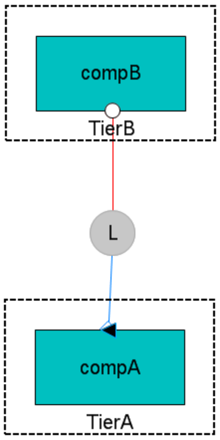
**Computing Conformance: Strategy #1.** When adopting Strategy #1, the conformance visitor does the following:

- Push non-conforming connections from the as-built to as-designed architecture;
- Ignore new top-level elements in the as-built view;
- Ignore component sub-structure in the as-built view;
- Add missing connections from the as-built view to the as-designed view, and represent them in terms of the as-designed architecture. For instance, if the as-built view has a connector between components $A'$ and $B'$, that match the as-built components $A$ and $B$, the analysis adds a connector between $A$ and $B$.

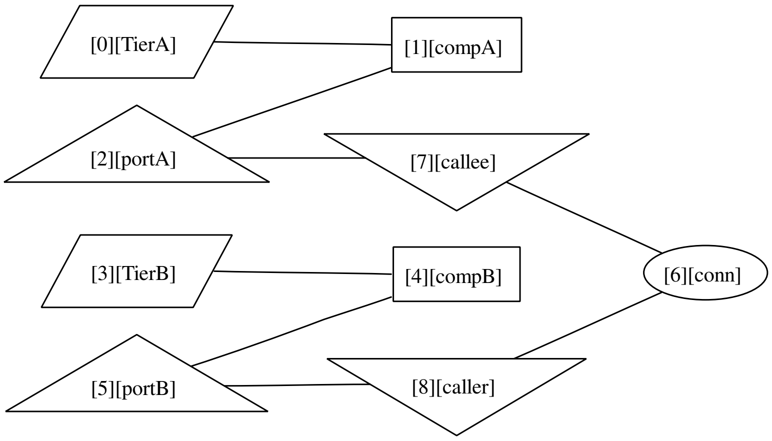
**Post-Processing Step.** The post-processing step requires building the adjacency matrices from the C&C views that once the hierarchical information has been matched. The adjacency matrices are built by creating a node for each Component, Port, Connector, Role, and Group. The effect of this adjacency matrix is to flatten the hierarchical graph. See Figure 5.4. Given elements $a$ and $b$, there is an edge from $\mathsf{nodeOf}(a)$ to $\mathsf{nodeOf}(b)$, for any of the following:

- Component $a$ has Port $b$;
- Connector $a$ has Role $b$;
- Group $a$ has Component $b$;
- Group $a$ has Connector $b$;
- Component $a$ has Representation $b$;
- Representation $a$ has Group $b$;

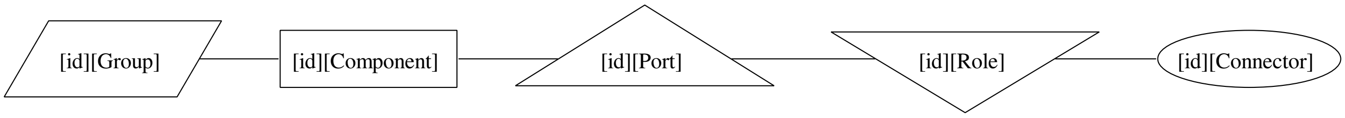
[4] The Reflexion Models approach [54] adopts Strategy #2: if a node is in the as-built model, but not in the as-designed model, it is automatically added to the as-designed model.




(a) Acme System




(b) Converting Acme System to graph.




(c) Graph legend.

Figure 7: Converting an Acme system to a graph to compute its adjacency matrix.

- Port $a$ has an Attachment to Role $b$;
- Outer Port $a$ has a Binding to inner Port $b$; etc.

**Reporting Conformance Results.** We report and visually represent the conformance results on the as-designed architecture, as follows:

- **Element Property:** we define on each Acme element a syncStatus property. The value of that property is assigned based on the match code;
- **System Property:** we define on the Acme System a number of conformance-related properties. For instance, a numMissing property tracks the number of components that are in the as-built architecture, but not in the as-designed architecture;
- **Visualization Variants:** we use the notion of "variants" to decorate each architectural element based on the value of the syncStatus property. Mehra et al. also highlight graphically their diagram differences [50];
- **Rules:** We use Acme heuristics (rules) to produce Acme errors based on the value of the System-level properties [51].

## 5.5 Illustrative Example

We illustrate the results of checking the conformance of the CourSys system.

**As-Designed Architecture.** The as-designed architecture is shown in Figure 8.

**As-Built Architecture.** As discussed in Section 3, the CourSys as-built architecture was extracted automatically. As is to be expected, the as-built architecture is more detailed.

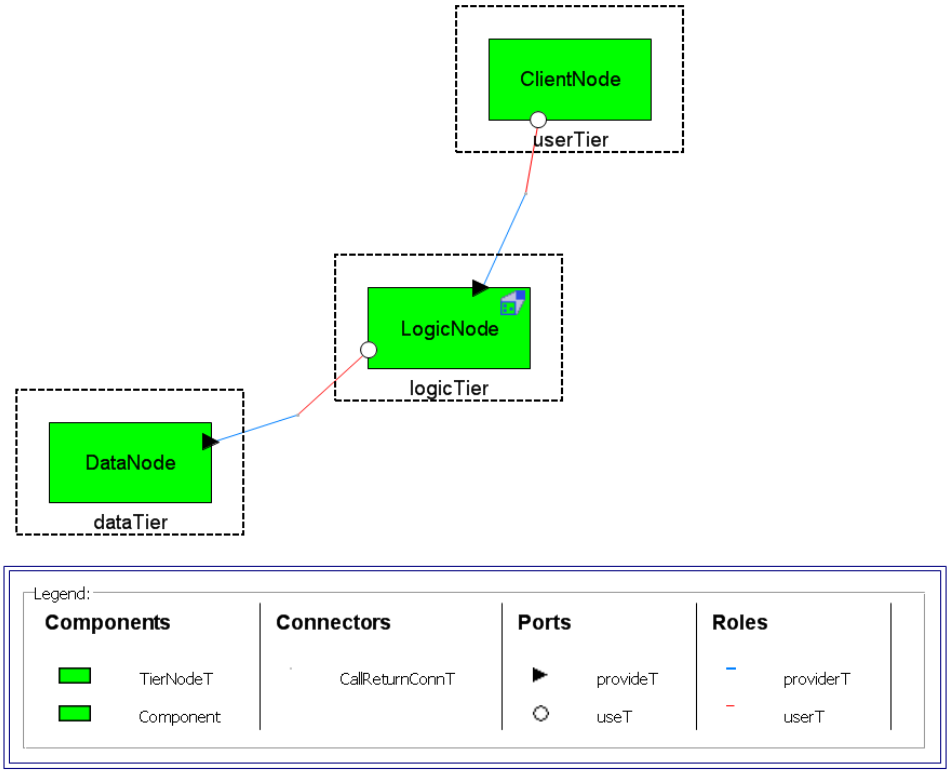



Figure 8: CourSys as-designed architecture in Acme.

**Comparison Results.** The results of the structural comparison between the as-built and the as-designed architectures are shown in Figure 10. These results were obtained without having the end-user manually force any matches between view elements.

The tool shows the structural differences by overlaying icons on the affected elements in each tree (See Figure 9). The tool helps the user understand the mapping as follows: if an element is renamed, the tool automatically selects and highlights the matching element in the other tree. For inserted or deleted elements, the tool automatically selects the insertion point, by navigating up the tree until it reaches a matched ancestor. The tool shows in bold a node if it detects differences in its subtree. Finally, the tool can generate a report with the match results.

(a) (b) (c) (d)

Figure 9: Figure 9(a) indicates a *match*; Figure 9(b) indicates a *rename*; Figure 9(c) indicates an *insertion*; and Figure 9(d) indicates a *deletion*.

We interpret the results of Figure 10 as follows:

- **Renames:** the tool successfully detected many renames. For instance, the `dataTier` group in the as-designed view corresponds to the `data` domain in the as-built view. Similarly, the `DataNode` component is mapped to the `objData` component. Inside `LogicNode`'s substructure in the as-designed view, `Logging` is mapped to a `log` component in the as-built view.
- **Inserts/Deletes:** the tool successfully detected many insertions. For instance, the `objData` component in the as-built view has an additional port `objLogic`, and additional substructure `repIData`, which contains a `state` tier, and components `course`, `student`, etc. The as-built view has an additional connector, named `objDataobjLogic_objLogicILogic`. Finally, the as-built view has an additional component `lock` inside `objLogic`'s substructure.

**Conformance Checking Results.** Figure 11 shows the conformance checking results, displayed on the as-designed architecture. In particular, it highlights the additional port on the `DataNode`, that is attached through a connector to the `LogicNode`.

Note, the appropriate domain links in the ownership domain annotations could prohibit that communication. But in this case, we relaxed the domain links for illustrative purposes.

**Summary: We use our existing tree-to-tree correction algorithm that detects inserts, deletes, renames and restricted moves, and supports forcing and preventing matches. A conformance visitor traverses the common-supertree produced by the algorithm, and propagates conformance findings from the as-built to the as-designed architecture. In particular, it pushes non-conforming connections from the as-built to as-designed architecture, and ignores a few details in as-built view, e.g., component sub-structure. The conformance results are represented in terms of the as-designed architecture, i.e., the inserted elements use the names from the as-designed view.**

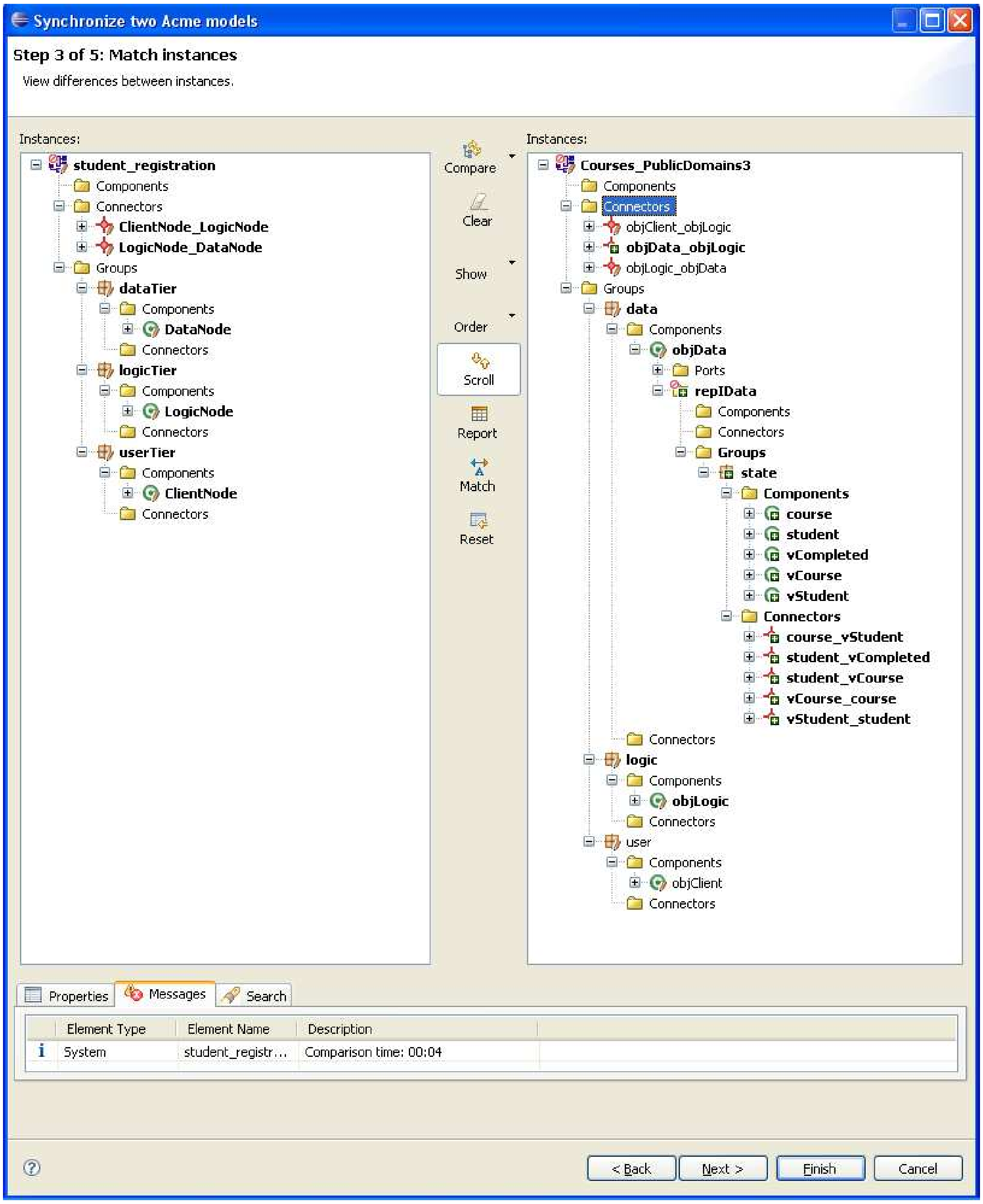



Figure 10: CourSys structural comparison between the as-built and the as-designed architectures.

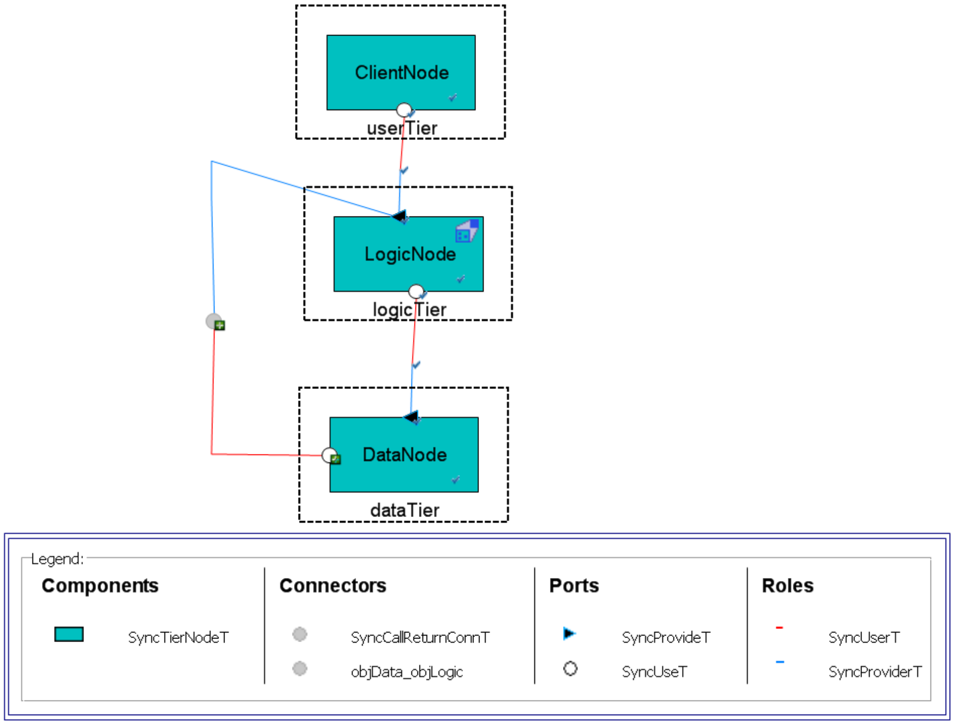



Figure 11: Conformance results between the CourSys as-designed and as-built architecture.

# 6 Measuring Structural Conformance

Our design-intent based analysis for checking and measuring architectural conformance has three stages: the *Annotation Stage*, the *Conformance Stage*, and the *Post-Synchronization Stage*. Different measurements are possible in each stage.

We follow the approach recommended by McGarry et al. [47], and the pattern of ISO/IEC 15939, to define the metrics. For each metric, we layout the definition of: 1) *base measures*; 2) show the calculations for any *derived measures*; 3) describe how to interpret those measures, i.e., *indicators*; and 4) identify the overall *information product*, as the business evaluation of the value or the risk.

## 6.1 Annotation Stage

In this stage, the following measurements are possible:

### 6.1.1 Percentage of Annotated Program

This measure includes the percentage of the program that was annotated, excluding any libraries that the program uses.

- **Base Measures:** 1) Total number of reference types in the program. An initial measure can only the number of field declarations of reference type, and not include method parameters and local variables of reference type; 2) Number of Reference Types (whether field declarations, local variables, or method parameters) in the program that have ownership domain annotations.
- **Derived Measure:** Percentage of Annotated Reference Types.
- **Indicator:** The larger the percentage, the more assurance we have that the extracted architecture faithfully reflects the actual system's execution architecture. Ideally, this value should increase over time.

### 6.1.2 Percentage of Annotated External Libraries In Use

Our annotation-based system allows annotating portions of any library that the program uses in external files [2]. For instance, annotating the CourSys program required creating 8 external files for annotating Java Standard Library classes (and interfaces), such as `java.io.BufferedReader`, `java.io.StreamTokenizer`, `java.lang.Object`, `java.lang.String`, `java.util.AbstractList`, `java.util.ArrayList`, and `java.util.Iterator`.

- **Base Measures:** 1) *Total number of reference types in external libraries*: An initial measure can include only the number of field declarations of reference type. Method parameters and local variables need not be initially included. 2) *Number of reference types in that have an ownership domain annotation stored in an external file*; and 3) *Number of Virtual Fields:* Short of annotating the entire Java Standard Library, our annotation-based system allows defining 'virtual" or "ghost" [26] fields in the external annotation files. A virtual field is a promise to soundly represent references which are internal to an abstraction that is not annotated. But if a virtual field is missing or incorrect, the extracted architecture may be missing some objects and relations that exist at runtime. For instance, annotating the CourSys program required defining an 'virtual" field so that the `ArrayList` has a field reference to the list element.
- **Derived Measure:** Percentage of Annotated Library.
- **Indicator:** The larger the percentage, the more assurance we have that the extracted architecture faithfully reflects the system's runtime architecture.

### 6.1.3 Residual Ownership Type Errors

The number of type errors that remain in the annotated programs, is valuable information to help understand what the final result is — in terms of confidence that the final annotated program actually satisfies the ownership domains type system.

$$\frac{\text{TARP} + \text{TARRL}}{\text{TRTP} + \text{TRTL} + \text{TTEP}}$$

| | |
|---|---|
| $TARP$ | Total No. of Annotated References in Prog. |
| $TARL$ | Total No. of Annotated Reachable References in Libs. |
| $TRTP$ | Total No. of Ref. Types in Prog. |
| $TRTL$ | Total No. of Reachable Ref. Types in Libs. |
| $TTEP$ | Total No. of Remaining Type Errors in Prog. |

### 6.1.4 Annotation Quality

A measure of the quality of the annotations is more meaningful that the raw number of annotations. For instance, objects marked with the `shared` annotation may be aliased globally, and little reasoning can be done about those references — except that they may not alias non-`shared` references.

`shared` references are often used to interoperate with existing libraries, legacy code and static fields, all of which may refer to aliases that are not confined to the scope of any object instance. In most other cases, a `shared` annotation is not very meaningful, and must be avoided. Thus one measure could be to count the percentage of shared annotations.

Similarly, making an object fully encapsulated is considered to be a high quality annotation, e.g,. by declaring a private domain `owned`, and marking its field as `owned`. Instance encapsulation avoids representation exposure, and eliminates a source of bugs. In fact, a popular code quality tool, FindBugs, warns about cases of suspected representation exposure, and it is precisely these kinds of mistakes in a program that ownership types prevent.

Similarly, public domains are high-quality annotations. Another measure can include the number of declared public domains in the program, and the percentage of objects that are annotated to be inside those public domains.

- **Base Measures:** 1) *Percentage of `shared` annotations*; 2) *Percentage of `lent` annotations*; 3) *Percentage of `unique` annotations*; 4) *Percentage of `owned` annotations*; 5) *Percentage of annotations that use public domains.*
- **Derived Measure:**
- **Indicator:** these numbers must be within acceptable limits. For instance, in a previous case study, we obtained the following numbers. Field annotations were broken down as follows: 45% as `owned`, 34% as `shared`, 20% as `domain parameters` and 1% as other annotations. Variable and method parameter declarations were broken down as follows: 69% as `lent`, 14 % as `shared`, 16% as `domain parameters` and 1% as other annotations [3].

  **Remarks.** These metrics are not yet fully implemented.

## 6.2 Conformance Stage

In this section, we measure the structural conformance of an implementation to its execution architecture.

### 6.2.1 First Pass Metrics

The first pass is used to synchronize the strictly hierarchical information, and produces the common supertree. The metric computed at the end of this phase is computed by a tree traversal of the common supertree – ignoring renames, substructure insertion, as discussed below. The metric thus consists of a *weighted edit distance* between the as-designed and the as-built architecture when they are both represented as graphs.

**Renames.** We assume that, in general, renames do not count against structural conformance, because one of the strengths of our approach is that it detects renames, compared to other tools, e.g., ArchDiff [14].

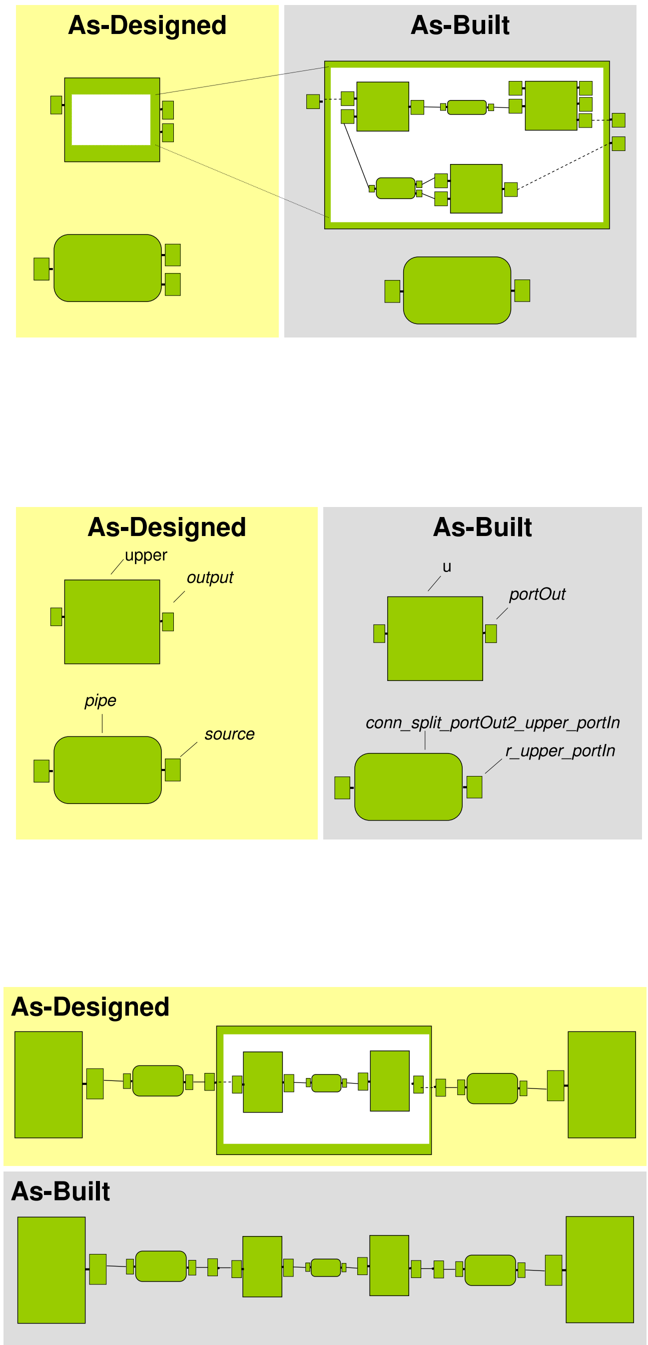
**Inserts/Deletes.** Whether or not an insert or a delete is important for conformance is based on the type of element that is inserted or deleted. We assume that the following do not count against conformance:
- If the as-built architecture shows additional substructure for a given component — which appears as an inserted component Representation, then the additional detail does not count against conformance;
- If the as-designed architecture does not specify some information that exists in the implementation, such as required and provided method signatures, this information can be excluded from the comparison to avoid false positives. Typically, this information appears as an inserted RequiredMethod or ProvidedMethod. Adding structural information to the as-designed view improves the match precision.

**Type Changes.** During synchronization, the type of a Port or a Role may need to be modified, to be compatible with the allowed "connection patterns": e.g., a Port of type provideT may not be connected to a Role of type userT. Types are not currently represented in the tree-structured data. So a type mismatch does not count against structural conformance at this point. The Post-Synchronization Stage, discussed below, will account for these type mismatches.

**Metric Computation.** The metric is computed by traversing the common super tree, examining the status of each node, and excluding the children of an inserted substructure.

### 6.2.2 Second Pass Metrics

The first pass synchronizes the strictly hierarchical information. The second pass knows the mapping between the nodes in the two graphs. So, the second pass metric focuses on correcting the connections, and can be considered a difference between two graph adjacency matrices.




(c) Replacing a component with its representation in the as-built architecture.

Figure 12: A few possible differences between the as-built and the as-designed architecture.

**Metric Computation.** In the following discussion, we use standard mathematical matrices. A matrix with $m$ rows and $n$ columns is an $m$-by-$n$ matrix (written $m \times n$). The entry of a matrix $A$ that lies in the $i$-th row and the $j$-th column is written as $a_{i,j}$ and called the $i, j$ entry or $(i, j)$-th entry of $A$. Alternative notations for that entry are $A[i, j]$ or $A_{i,j}$. We note the row first, then the column.

We use $|a|$ to denote the standard notion of *absolute value* of $a$, defined as:

$$|a| = \begin{cases} a, & \text{if } a \geq 0 \\ -a, & \text{if } a < 0. \end{cases}$$

**Absolute Difference.** Given two $m$-by-$n$ matrices $A$ and $B$, we define their *absolute difference* $|A - B|$ as the $m$-by-$n$ matrix computed by subtracting the corresponding elements and then obtaining the absolute value of the difference:

$$\begin{aligned} |\mathbf{A} - \mathbf{B}| &= |(a_{i,j})_{1 \leq i \leq m;\, 1 \leq j \leq n} - (b_{i,j})_{1 \leq i \leq m;\, 1 \leq j \leq n}| && (1) \\ &= (|a_{i,j} - b_{i,j}|)_{1 \leq i \leq m; 1 \leq j \leq n} && (2) \end{aligned}$$

For example:

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix} | - | \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} |1-0| & |0-0| & |1-1 \\ |1-0| & |0-1| & |0-0| \\ |1-1| & |0-1| & |1-1| \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

**Remark.** The matrices correspond to the graph adjacency matrices previously computed in Section 5.4. A value of 1 denotes the presence of an edge between two nodes, and a value of 0 denotes its absence.

**Core Difference.** Before we can compute the Core Difference between the as-designed and the as-built adjacency matrices, we have to take into account the mapping table between the nodes. The table is produced by the earlier hierarchical synchronization. Conceptually, the effect of the mapping table is to re-order the entries in the as-built matrix so that the comparison is meaningful. We then compute the absolute difference of the resulting matrix to the as-designed one, and add up the entries to obtain the Core Difference:

$$\begin{aligned} \Sigma|\mathbf{A} - \mathbf{B}| &= \Sigma|(a_{i,j})_{1 \leq i \leq m;\, 1 \leq j \leq n} - (b_{i,j})_{1 \leq i \leq m;\, 1 \leq j \leq n}| && (3) \\ &= \Sigma(|a_{i,j} - b_{i,j}|)_{1 \leq i \leq m; 1 \leq j \leq n} && (4) \end{aligned}$$

**Core Ratio.** The Core Ratio is the ratio of the sum of the entries over the total number of matrix entries:

$$\text{Core Ratio} = (\Sigma(|a_{i,j} - b_{i,j}|)_{1 \leq i \leq m; 1 \leq j \leq n}) \div (m \times n) \times (100\%) \qquad (5)$$

In the earlier example, the Core Ratio would be: $4 \div 9 \sim 44\%$.

**Residual Difference.** The Residual Difference accounts for any entries in the as-built adjacency matrix that are not in the as-designed matrix. The Residual Difference does not consider the value of the adjacency matrix entries (we emphasize this by showing . and X for the white entries in Figure 13, instead of 0 or 1 for the grey entries). Of course, not considering this value amounts to treating all the entries as 1, which is the worst case scenario of a fully connected graph. From a security standpoint, this is the conservative assumption; if a component $A$ communicates with component $B$, and $B$ with $C$, then $A$ could communicate with $C$ transitively.

Ideally, following Strategy #2 discussed earlier, the Residual Difference should close to zero, when the as-designed architecture and the as-built architecture would roughly same number of elements (nodes) — even though those nodes may be connected differently.

The pseudo-code for computing the Core Difference and the Residual Difference is in Figure 14.

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 1 | 1 |
| B | 1 | 0 | 1 | 1 | 1 |
| C | 1 | 1 | 0 | 1 | 1 |
| D | 1 | 1 | 1 | 0 | 1 |
| E | 1 | 1 | 1 | 1 | 0 |

As-Designed
L x L

| | A' | B' | C' | D' | E' | F' | G' |
|---|---|---|---|---|---|---|---|
| A' | 1 | 1 | 0 | 1 | 0 | . | . |
| B' | 0 | 1 | 1 | 1 | 1 | . | . |
| C' | 0 | 1 | 0 | 0 | 0 | . | X |
| D' | 1 | 1 | 1 | 0 | 0 | . | . |
| E' | 1 | 0 | 1 | 1 | 1 | . | . |
| F' | X | . | . | . | . | X | . |
| G' | X | . | . | . | . | . | . |

As-Built Adj. Matrix
R x R

| | C | E | A | D | B |
|---|---|---|---|---|---|
| C | 1 | 1 | 0 | 1 | 0 |
| E | 0 | 1 | 1 | 1 | 1 |
| A | 0 | 1 | 0 | 0 | 0 |
| D | 1 | 1 | 1 | 0 | 0 |
| B | 1 | 0 | 1 | 1 | 1 |

As-Built Matrix
Based on mapping

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 0 | 0 | 0 | 1 |
| B | 1 | 1 | 1 | 1 | 0 |
| C | 0 | 0 | 1 | 1 | 1 |
| D | 1 | 0 | 1 | 0 | 1 |
| E | 1 | 1 | 0 | 1 | 1 |

Reordered
As-Built Matrix

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 1 | 1 | 1 | 0 | 0 |
| B | 0 | 0 | 0 | 0 | 1 |
| C | 1 | 1 | 1 | 0 | 0 |
| D | 0 | 1 | 0 | 0 | 0 |
| E | 0 | 0 | 1 | 0 | 1 |

Core Difference
= 10
Core Ratio
= 10 / 25

| L | R |
|---|---|
| A | C' |
| B | E' |
| C | A' |
| D | D' |
| E | B' |

Mapping Table

Residual Difference = ...
Conformance Metric = 1 - (Core Difference + Residual Difference) / (L x R)

Figure 13: Graphical illustration of the Conformance Metric, based on the graph adjacency matrices, of the as-designed and the as-built architectures.

```
double coreDiff = 0; /* Core Difference */
double resDiff = 0; /* Residual Difference */
Graph graphL;
Graph graphR;
Hashtable mapRtoL; /* Map elements on the R to elements  on the L */
int numL = graphL.numberOfVertices;
int numR = graphR.numberOfVertices;

for (int ii = 0; ii < numR; ii++) {
    Vertex ndR = graphR.vertices[ii];
    Vertex ndL = mapRtoL.get(ndR);
    if ( ndL != null ) {
        for (int jj = ii + 1; jj < numR; jj++) {
            Vertex tndR = graphR.vertices[jj];
            Vertex tndL = mapRtoL.get(tndR);
            if ( tndL != null ) {
                if ( getAdjMatrixR(ndR,tndR) != getAdjMatrixL(ndL,tndL) ) {
                    coreDiff++;
                }
            }
            else {
                resDiff++;
            }
        }
    }
    else {
        resDiff++;
    }
}
```

Figure 14: Pseudo-code for computing the Conformance Metric.

**Conformance Metric.** The Conformance Metric accounts for both the Core Difference and Residual Difference. The higher this number is, the more the as-built extracted architecture conforms to the as-designed one. In the following, we assume that the as-designed view is on the left; and the as-built view is on the right. L refers to the dimension of the as-designed adjacency matrix; R is that of the as-built view.

$$\text{Conformance Metric} = (1 - (\text{Core Difference} + \text{Residual Difference})/(\text{L} \times \text{R})) \times 100\% \tag{6}$$

**Illustrative Example.** The conformance metrics for the earlier CourSys example are in Table 1.

Table 1: CourSys conformance metrics.

| | **Core Difference** | **Residual Difference** | **Conformance Metric** | **Size L** | **Size R** |
|---|---|---|---|---|---|
| CourSys | 0 | 563 | 50% | 18 | 62 |

## 6.3 Post-Synchronization Stage

There are additional conformance metrics that can be measured, once the as-designed and the as-built architectures are synchronized.

### 6.3.1 Types and Styles

Supplementing the C&C view extracted from the implementation with architectural types and styles can uncover additional violations. The architect can further enrich the up-to-date architectural model with additional constraints, heuristics and properties [51].

During the Annotation Stage, we measure the number of type errors related to the ownership domain annotations. Similarly, in the Post-Synchronization Stage, we measure the number of violations of architectural types, styles, constraints and heuristics.

**Domain Links.** Domain links are policies between ownership domains. Acme does not have the notion of a first-class edge between two Groups. However, a group can have associated properties and rules with it. For example, to specify a link between two Groups, one possibility is to use a System property, which defines the domain links.

```
Property domainLinks : string = "logicTier -> dataTier";

group logicTier = {
}

group dataTier = {
}
```

### 6.3.2 Structural Constraints

There are several possible structural constraints that an architect can enforce on the architecture. Enforcing these constraints can help prevent *architectural drift* or *erosion* during software evolution [59], more effectively than the program, with or without annotations. In the unannotated program, changing the execution architecture is as simple as passing a reference to an object. The ownership annotations somewhat help. But a developer can still add communication paths by modifying domain links, declaring additional domain parameters and passing additional domain arguments at object allocation sites. Code reviews could audit such changes. For instance, in Figure 2, for `Data` to require the `logicTier` parameter looks suspicious.

If the extracted architecture reflects such architecture-modifying changes, it makes it easier to trigger an architecture review. The constraints can be enforced by a visual inspection of the extracted architecture. Or once the extracted architecture is converted to a C&C view in an ADL, the ADL can enforce several structural constraints.

Empirical evidence suggests that such policies are frequently needed. For instance, a study using JHotDraw mentioned that "a common architectural mistake [...] was to provide `Figure`s with a reference to the `Drawing` or the `DrawingView`. `Figure`s do not by default have any access to either [...] This prevents them from accessing information such as the size of the `Drawing`. However, [some students overcame this] by passing the view into the constructor of a `Figure`, which can then store and access this as required" [41].

Ownership domain annotations could enforce some constraints, but also require changing the code. For instance, using a method domain parameter instead of a class domain parameter can prevent a `Handle` from holding on to a `DrawingView` object that is passed to it [2]. But enforcing these constraints on the architecture does not require changing the annotations or the code. In addition, domain links treat all communication equally, forcing developers to add domain links. But a policy allow only "weak" references between `Model` and `View` to ensure that the "change propagation is the only link between the model and the views and controllers" [13, p. 127].

**Examples.** Predicates in the Acme ADL can enforce structural constraints, such as:

- Component instance X is never directly connected to Component instance Y:

```
forall comp1 : Component in self.COMPONENTS |
    forall comp2 : Component in self.COMPONENTS |
    connected(comp1, comp2) ->  ! (comp1 == X AND comp2 == Y);
```

- A Component of type X is never directly connected to a Component of type Y:

```
forall comp1 : Component in self.COMPONENTS |
    forall comp2 : Component in self.COMPONENTS |
        connected(comp1, comp2) -> !(declaresType(comp1, X) AND declaresType(comp2, Y));
```

- There are no components in Group X that communicate with any component in Group Y directly.

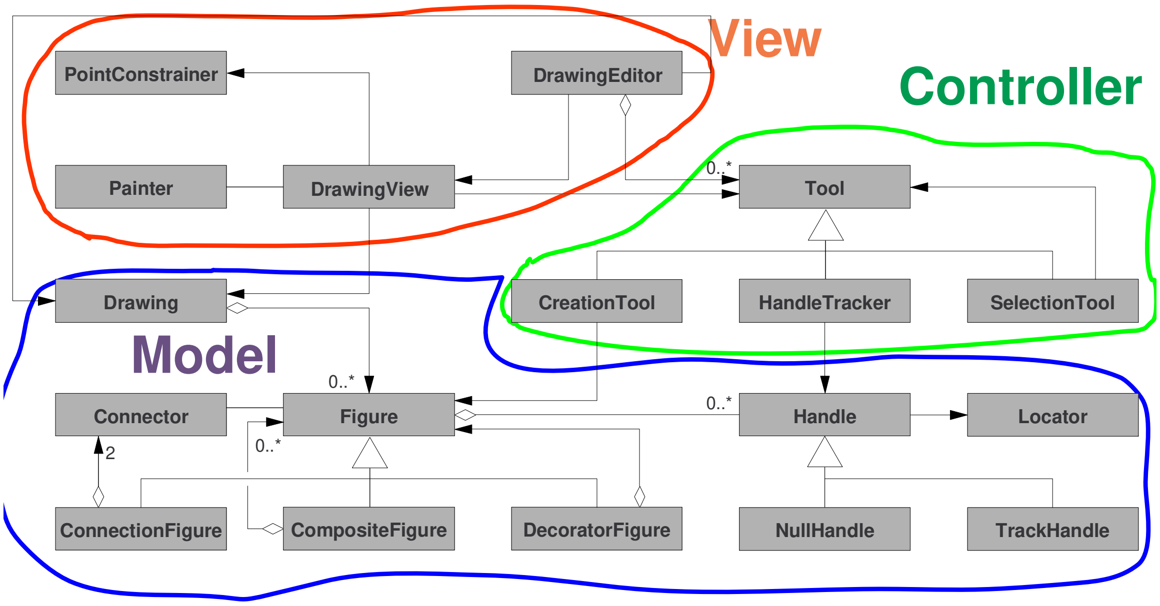



Figure 15: JHotDraw class diagram (Source: [61]).

# 7 Evaluation

In this section, we evaluate the approach on two extended examples.

## 7.1 JHotDraw

The first subject system is JHotDraw [29], a significant example in the object-oriented community. JHotDraw is open source, rich with design patterns [28], uses composition and inheritance heavily and has evolved through several versions. Version 5.3 has around 200 classes and 15,000 lines of Java.

**As-Designed Architecture.** JHotDraw's execution architecture was not documented. Since JHotDraw has been studied extensively, we were able to find many design artifacts on the web — although many are for older or newer versions, e.g., [27, 61, 38]. The class diagram in Figure 15 shows some of the core abstractions in JHotDraw. A widely cited article [38] discussed how JHotDraw followed the Model-View-Controller (MVC) design pattern [28]. JHotDraw is neatly organized into different packages. However, looking at the names of the packages does not indicate that JHotDraw follows the MVC pattern. In fact, all the core types are defined in one `framework` package.

We converted the diagram into an as-designed architecture that we documented in Acme (Figure 16). Of course, a static code architecture, such as the one in Figure 15, cannot be directly converted into an execution architecture. We were heavily inspired by the as-built architecture that the tool extracted from the annotated program. However, we explicitly did not add components from the *application model*, such as `UndoManager`, `StorageFormatManager`, etc., in the as-designed view. But we did include selected components from the *domain model*, such as `Figure`, `Handle`, etc.

We also modeled `Drawing` and `Figure` as one component in the as-built view. We knew from a previous case study that this was the case [1]. When we examined the extracted architecture, we were surprised that one of the core types in Figure 15, `Figure`, did not appear in the OOG. The extraction tool tracks the abstract objects and their associated types that are merged into a given visual object (See Figure 5). We used that information to determine that `Figure` and `Drawing` were merged in `Model`, and shown as `textFigure1:Drawing` in Figure 17.

This was because the base class implementing the `Drawing` interface, `StandardDrawing`, extends `CompositeFigure`. Thus a `Drawing` *is-a* `Figure`, to enable nesting a `Drawing` inside another `Drawing`. Even though this fact was mentioned in the Version 5.1 Release Notes, it was still unexpected. In the `framework` package, interface `Drawing` did not extend interface `Figure`. In their tutorial, the JHotDraw designers explicitly asked developers to "not commit to the `CompositeFigure` implementation, since some applications need a more complicated representation" [27, Slide #16]. In the final version of the tool, combining `Drawing` and `Figure` into one component in the as-designed view will not be necessary, because the tool is being modified to scan object allocations, instead of field and variable declarations [1].

**Adding Ownership Domain Annotations.** We annotated JHotDraw without making any structural refactoring such as extracting interfaces, etc. Some changes were needed however to use our annotation system: e.g., extract a local variable from a new expression to add an annotation on the local variable, convert an anonymous class to a nested class to add domain parameters to it, etc. Additional details of the annotation process are available elsewhere [2].

**Extracting the As-Built Architecture.** Using our tool, we extracted the as-built architecture from the annotated program (See Figure 17), and represented it in an Acme C&C architecture (not shown), as discussed above.

**Checking Conformance.** Next, we ran the architectural differencing tool between the as-designed (Figure 16), and the as-built architecture (Figure 17). The results are shown in Figure 18.

We interpret the results of Figure 10 as follows:

- **Renames:** the tool successfully detected many renames. For instance, the `Command` component in the as-designed view is mapped to the `cmd` component in the as-built view;
- **Inserts/Deletes:** the tool successfully detected many insertions. For instance, the tool detected all the "application model" components, such as `fStorageFormatManager`, `myUndoManager`, etc.

Figure 19 shows visually the conformance results. This figure mostly shows the "positive assurance" of JHotDraw's as-built architecture. This result is unsurprising. The as-designed architecture is mostly the extracted as-built architecture, without the "application model" components. At least, most elements in the as-designed view are named differently than in the as-built view (except for the tiers).

**Measuring Conformance Metrics.** The conformance metrics for JHotDraw are in Table 2. Again, the relatively high conformance measure is consistent with our earlier explanation. It also seems intuitive that adding the "application model" components to the as-designed architecture would raise the conformance measure even higher.

Table 2: JHotDraw conformance metrics.

| | Core Difference | Residual Difference | Conformance Metric | Size L | Size R |
|---|---|---|---|---|---|
| JHotDraw | 5 | 9144 | 83% | 159 | 334 |

## 7.2 HillClimber

By many accounts, JHotDraw is the brainchild of experts in object-oriented design and programming. In comparison, the second subject system, HillClimber, is another 15,000 line Java application that was developed by undergraduates. Our goal was to demonstrate that the approach works for programs that are not as well-designed as JHotDraw.

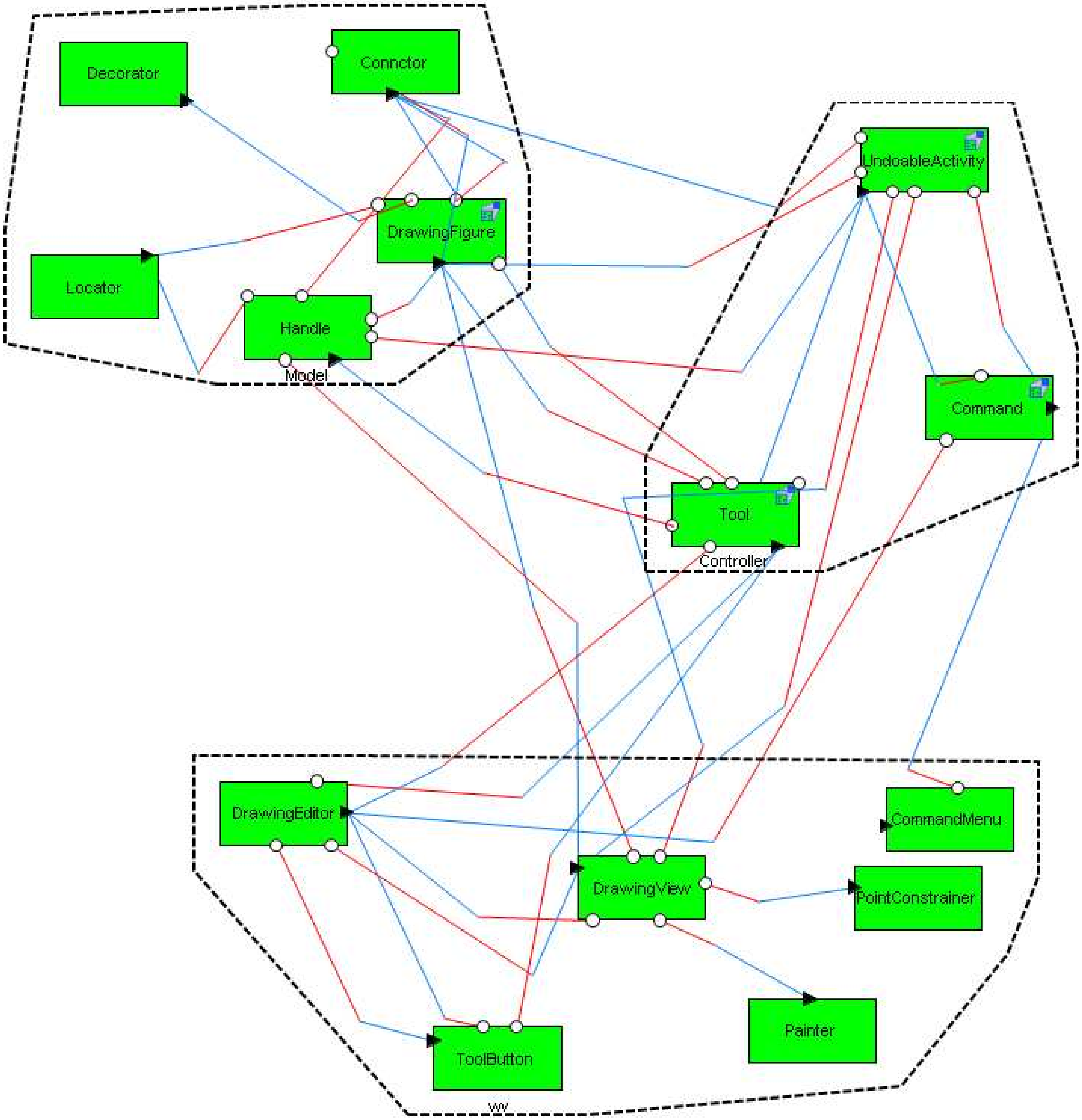



Figure 16: The JHotDraw as-designed architecture documented in Acme.

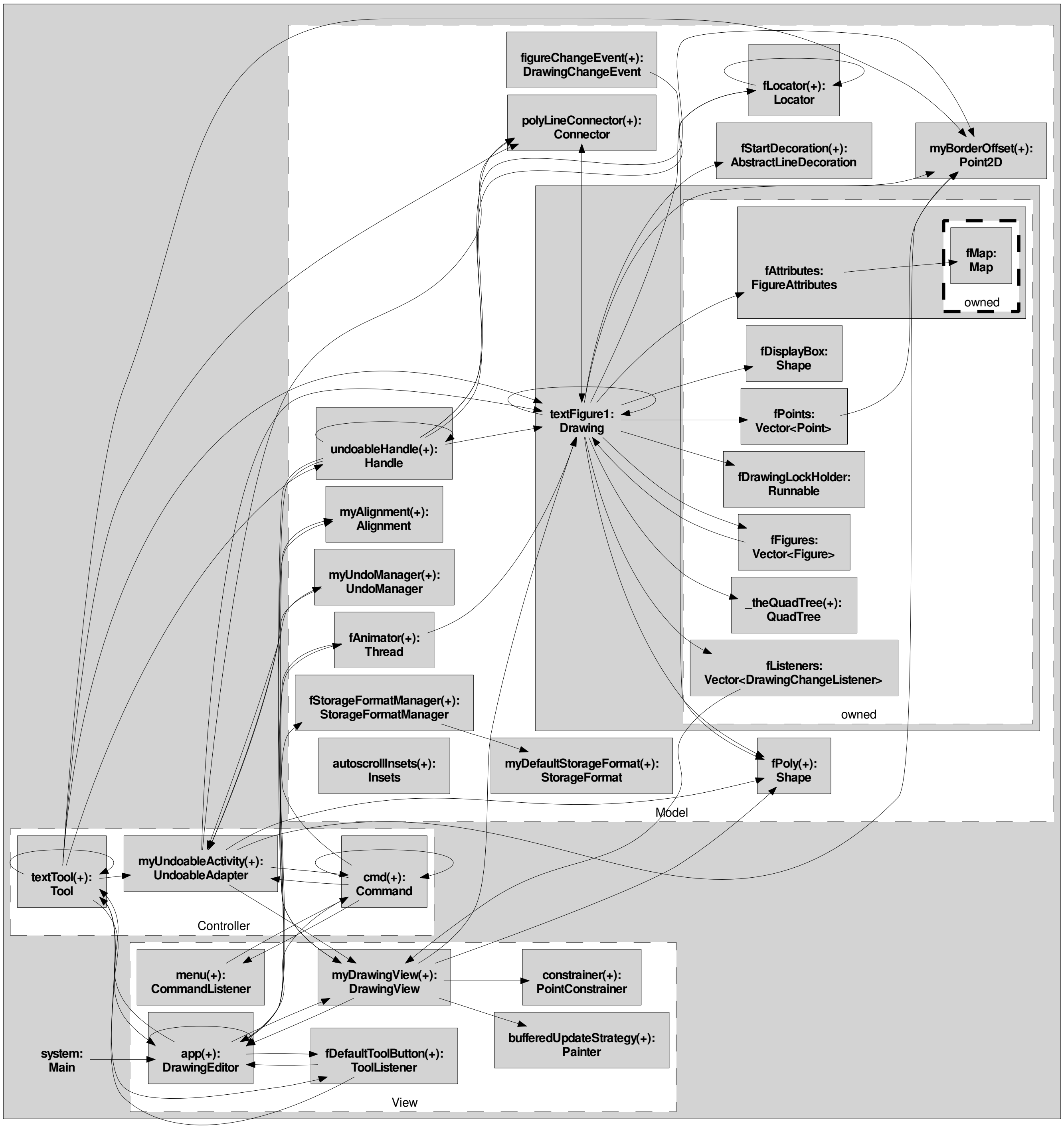



Figure 17: The JHotDraw extracted architecture.

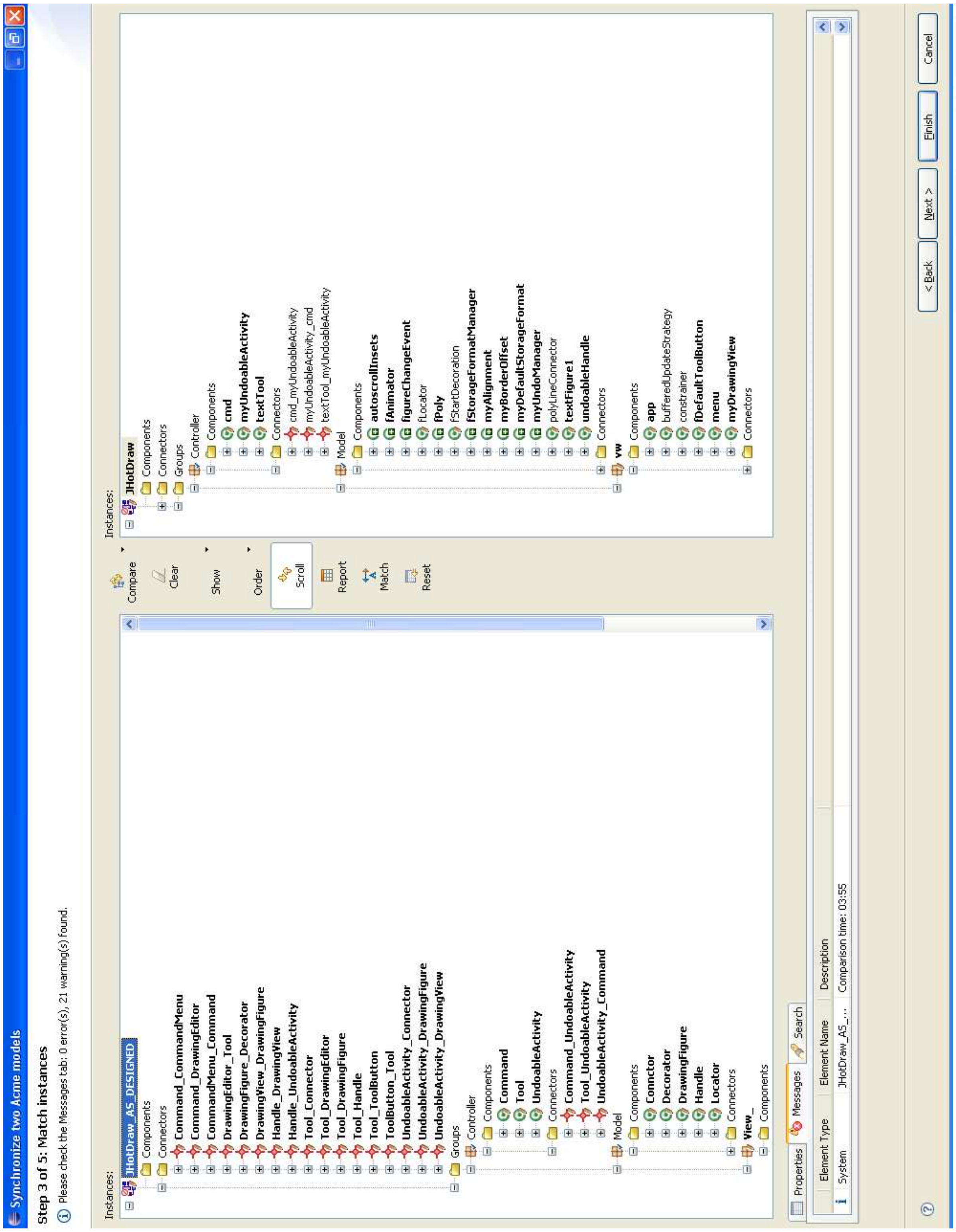



Figure 18: JHotDraw structural comparison between the as-built and the as-designed architectures.

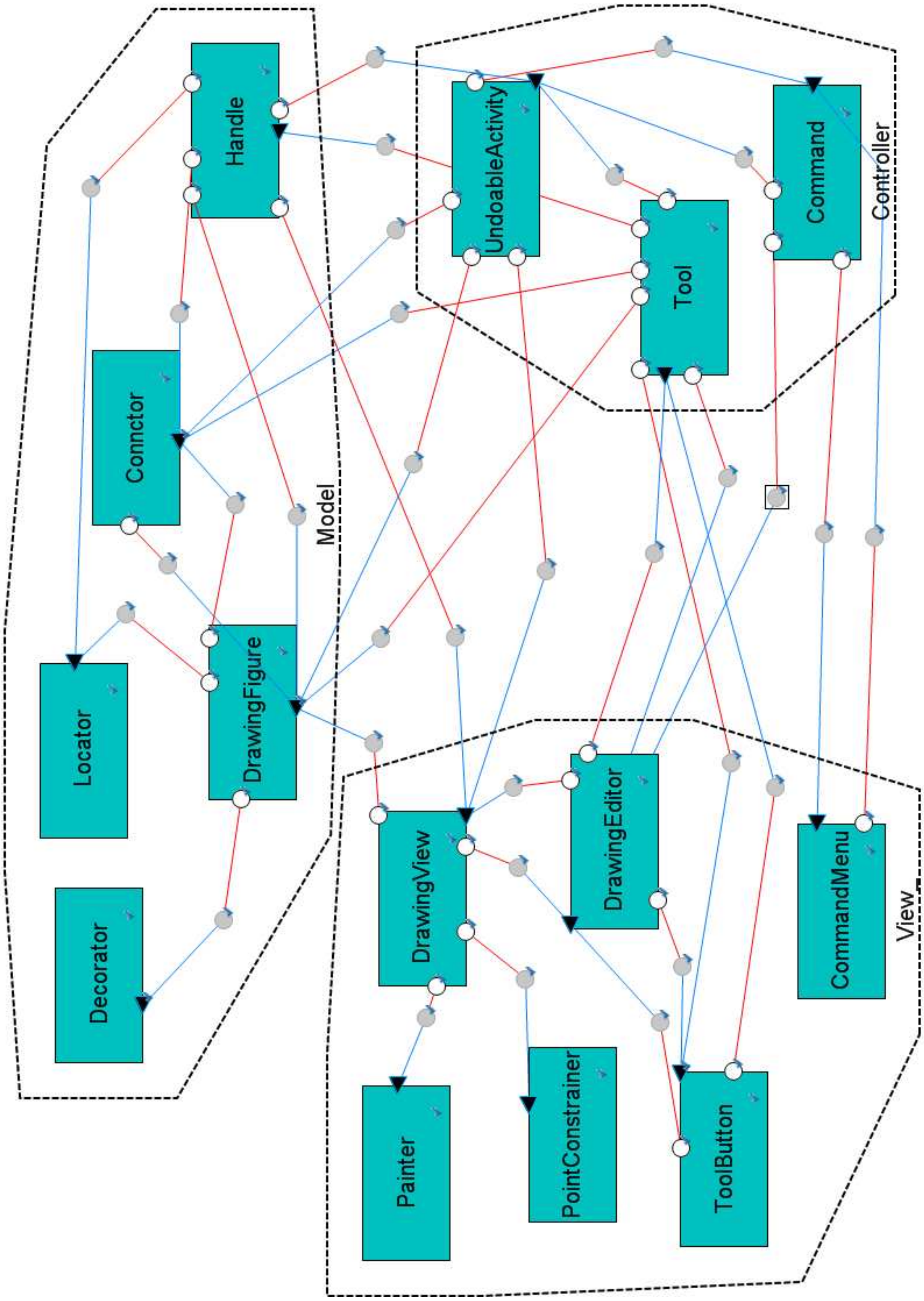



Figure 19: JHotDraw conformance results.

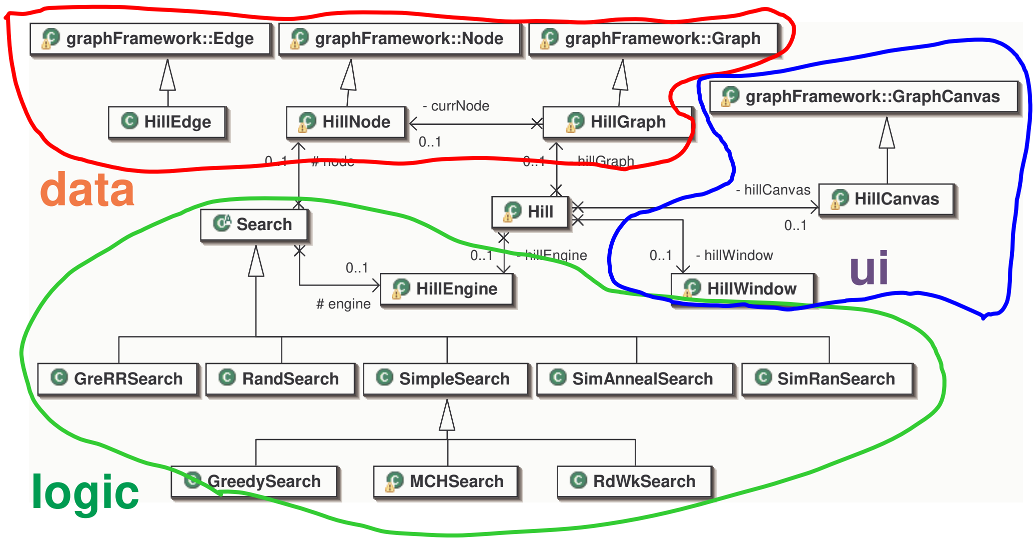



Figure 20: HillClimber class diagram.

**As-Designed Architecture.** In HillClimber, the application *window* uses a *canvas* to display *nodes* and *edges* of a *graph* to demonstrate algorithms for constraint satisfaction problems provided by the *engine*. The HillClimber UML class diagram, extracted from the implementation using Eclipse UML [58], is in Figure 20. The HillClimber as-designed architecture is in Figure 21.

**Adding Ownership Domain Annotations.** We previously discussed in details the process of annotating HillClimber [2].

**Extracting the As-Built Architecture.** The HillClimber extracted architecture is in Figure 22.

The extracted architecture in Figure 22 shows clearly the core HillClimber top-level objects, `window`, `canvas`, `engine` and `graph`. Similarly, the `Search` object in the `logicTier` domain merges many instances of sub-classes of class `Search` such as `MCHSearch`, `RandSearch`, etc.

The `CanvasMediator` object was introduced during a refactoring to decouple the code [2]. The `window` object merges several user interface objects such as dialogs.

We studied HillClimber with two sets of annotations. The earlier architecture showed a `dataTier` that was more cluttered [1]. We reduced the `dataTier` clutter, by changing several annotations to make more objects `owned` or `unique` [6]. For instance, we made `graph:HillGraph` own `heap:HillHeap`. We also made a few vectors `owned`, and ensured that the other references to them were `unique`, since they were passed linearly between objects [6]. In a few cases, we had to change the code to get the desired annotations, e.g., to return a copy of an internal list instead of an alias and avoid the representation exposure.

To reduce the number of top-level objects using logical containment, we also used public domains. Public domains group related objects by pushing them down the ownership tree, and removing them from the

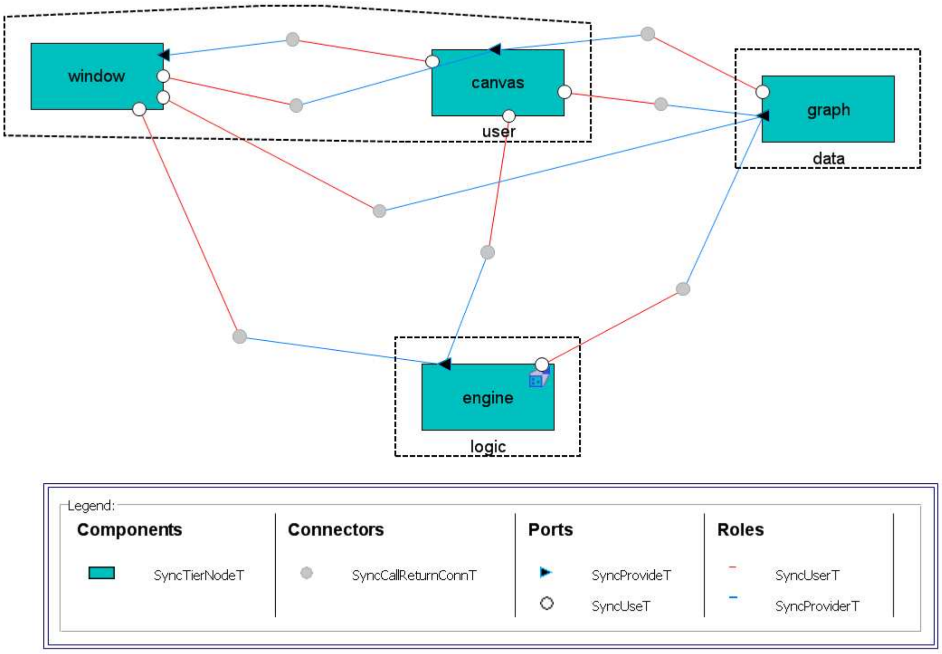



Figure 21: The HillClimber as-designed architecture.

top-level domains — while keeping them accessible to objects that can access the outer object (permission to access an object implies permission to access its public domains). For instance, in Figure 22, `randSearch` has a `heuristics` public domain with two array objects inside it, that object `heuristics` accesses.

The HillClimber as-built architecture represented in Acme is in Figure 23.

**Checking Conformance.** Figure 24 shows the results of the structural comparison. Note the additional edges between `engine` and `window` and `canvas`. Figure 25 shows graphically the results of the conformance check.

**Measuring Conformance Metrics.** The conformance metrics for HillClimber are in Table 3. The better annotations did reduce the size of the as-built graph slightly, but did not produce a higher value for the Conformance Metric. Perhaps, the metric should be made more precise to take the difference in the graph sizes as well.

Table 3: HillClimber conformance metrics.

| | **Core Difference** | **Residual Difference** | **Conformance Metric** | **Size L** | **Size R** |
|---|---|---|---|---|---|
| HillClimber | 0 | 4367 | 64% | 47 | 260 |
| HillClimber (now) | 0 | 4362 | 64% | 47 | 255 |

**Discussion.** The conformance metric for HillClimber is significantly lower than that of JHotDraw. This can be partly attributed to an as-designed architecture that has many fewer elements at the top-level than the as-built architecture. The recommendation in this case is either to: a) enrich the as-designed architecture; or b) keep fine-tuning the annotations to reduce the number of top-level elements in the as-built architecture.

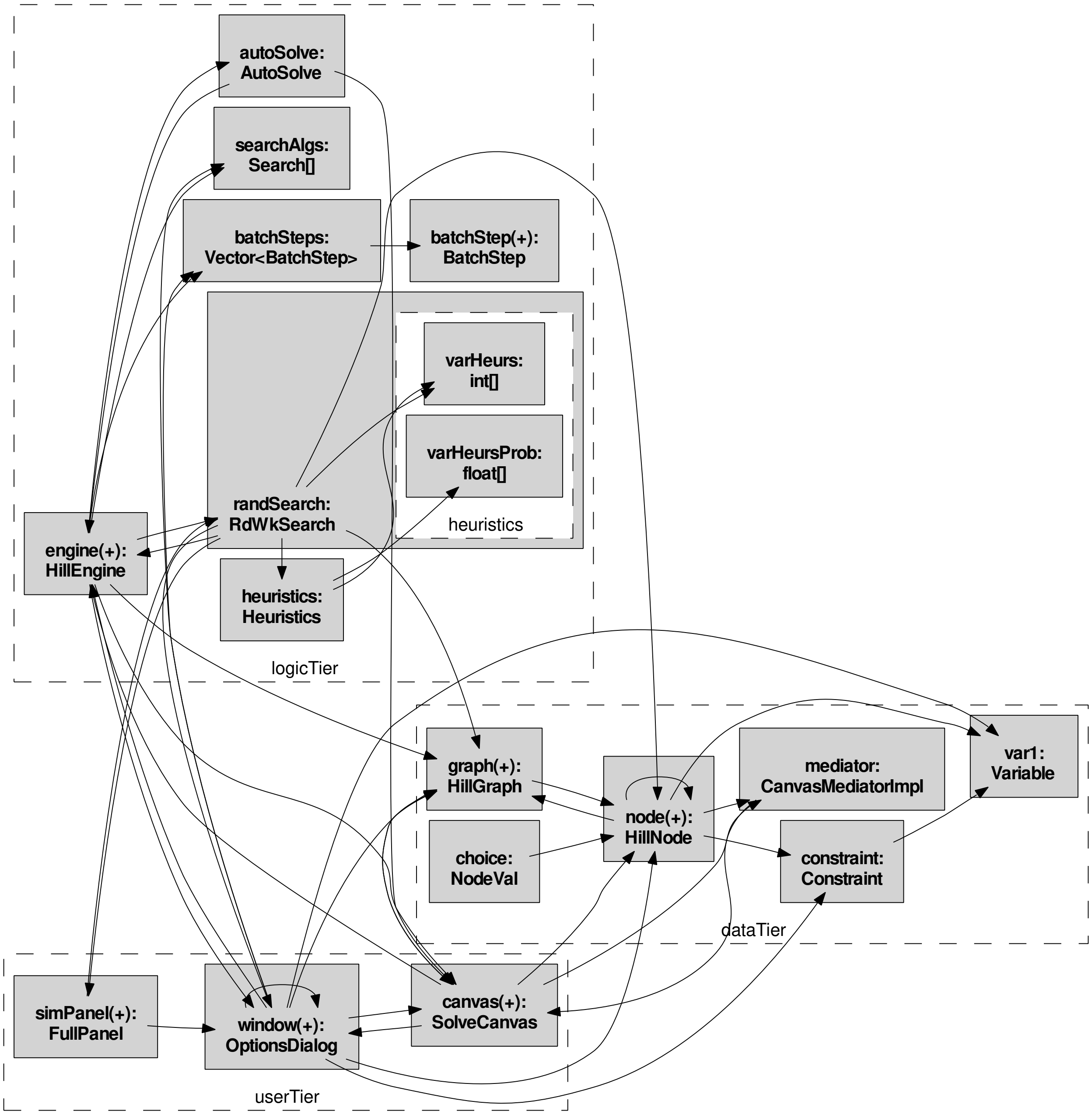



Figure 22: Top-level HillClimber extracted architecture.

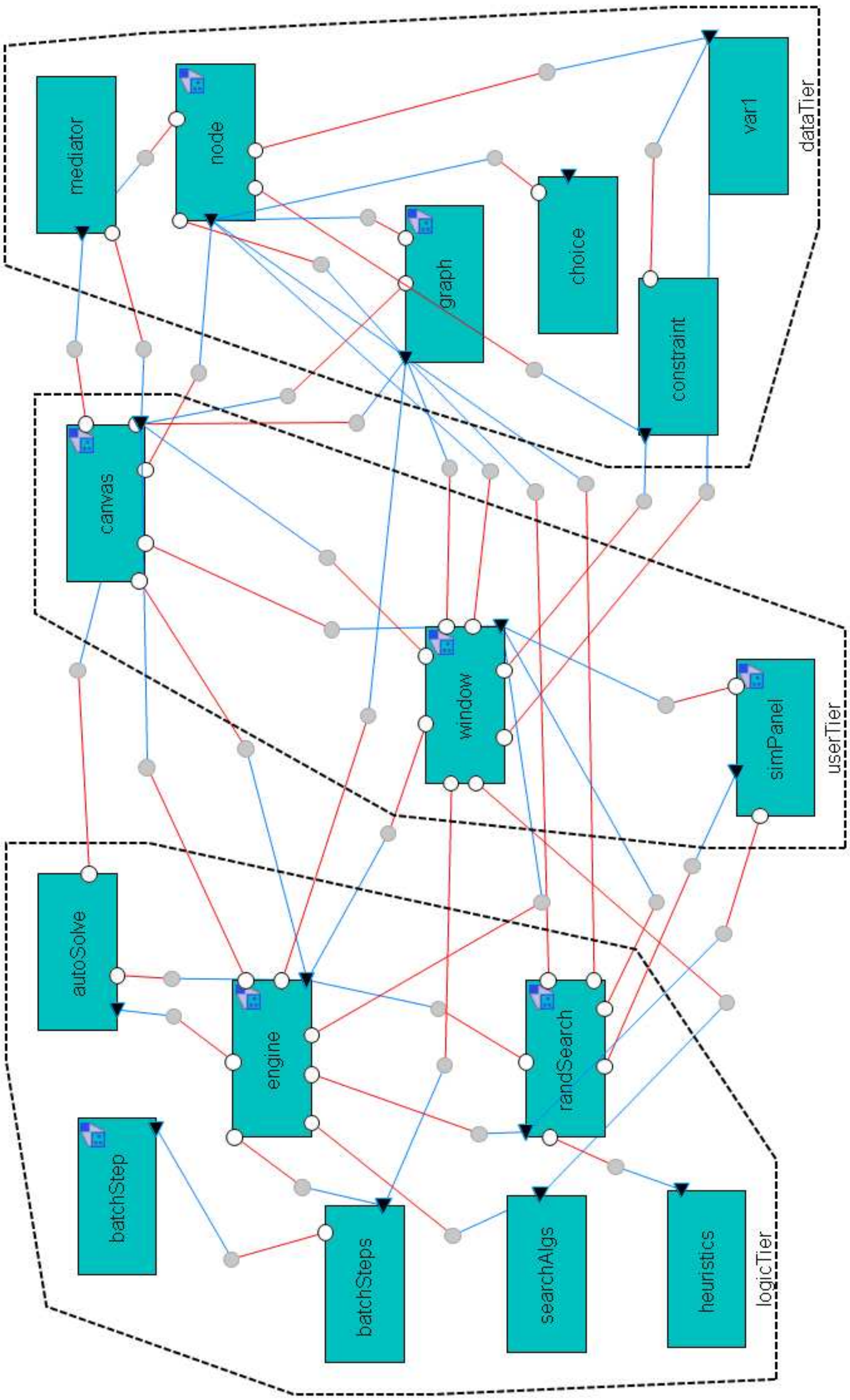



Figure 23: The HillClimber as-built architecture.

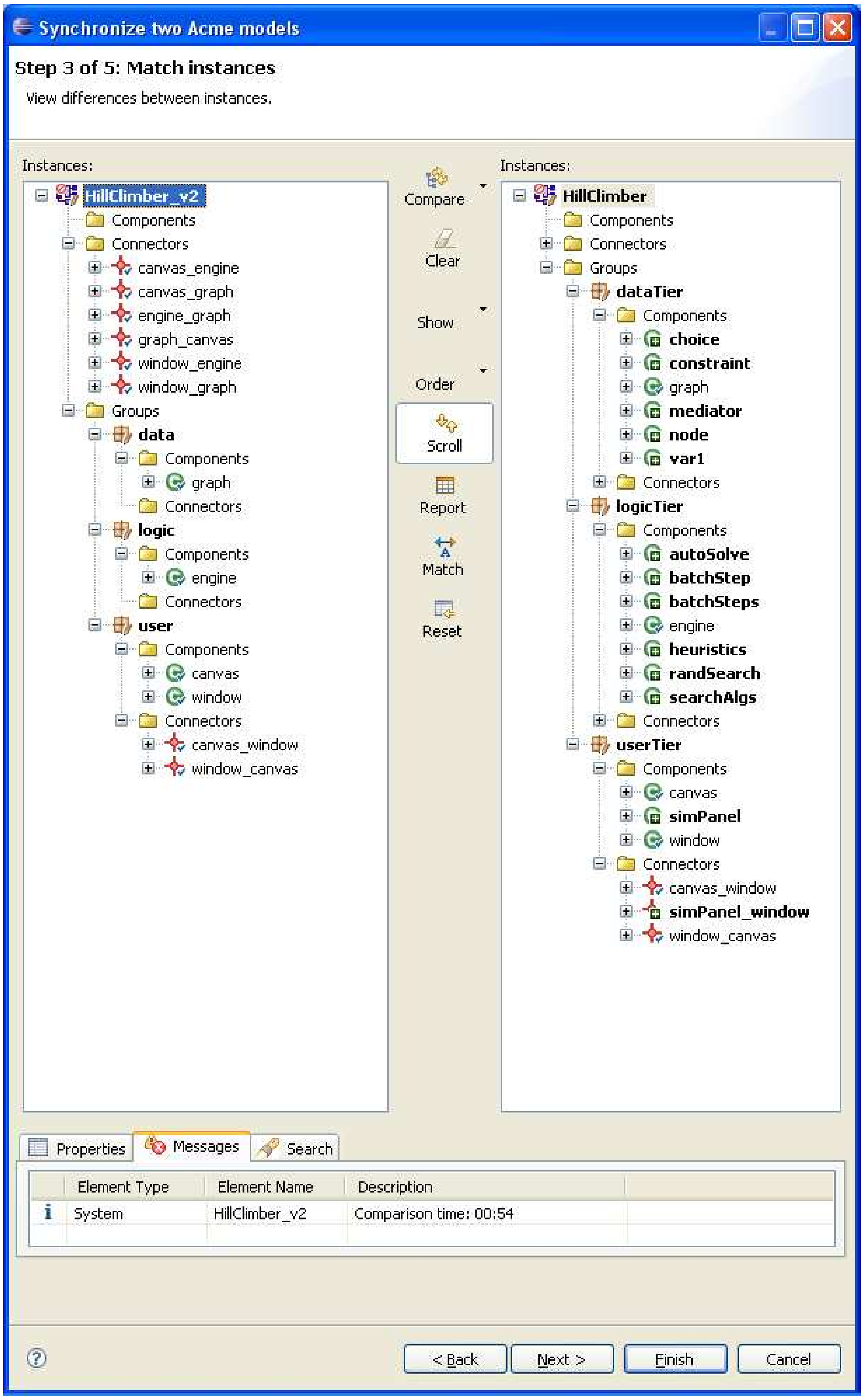



Figure 24: HillClimber structural comparison between the as-built and the as-designed architectures.

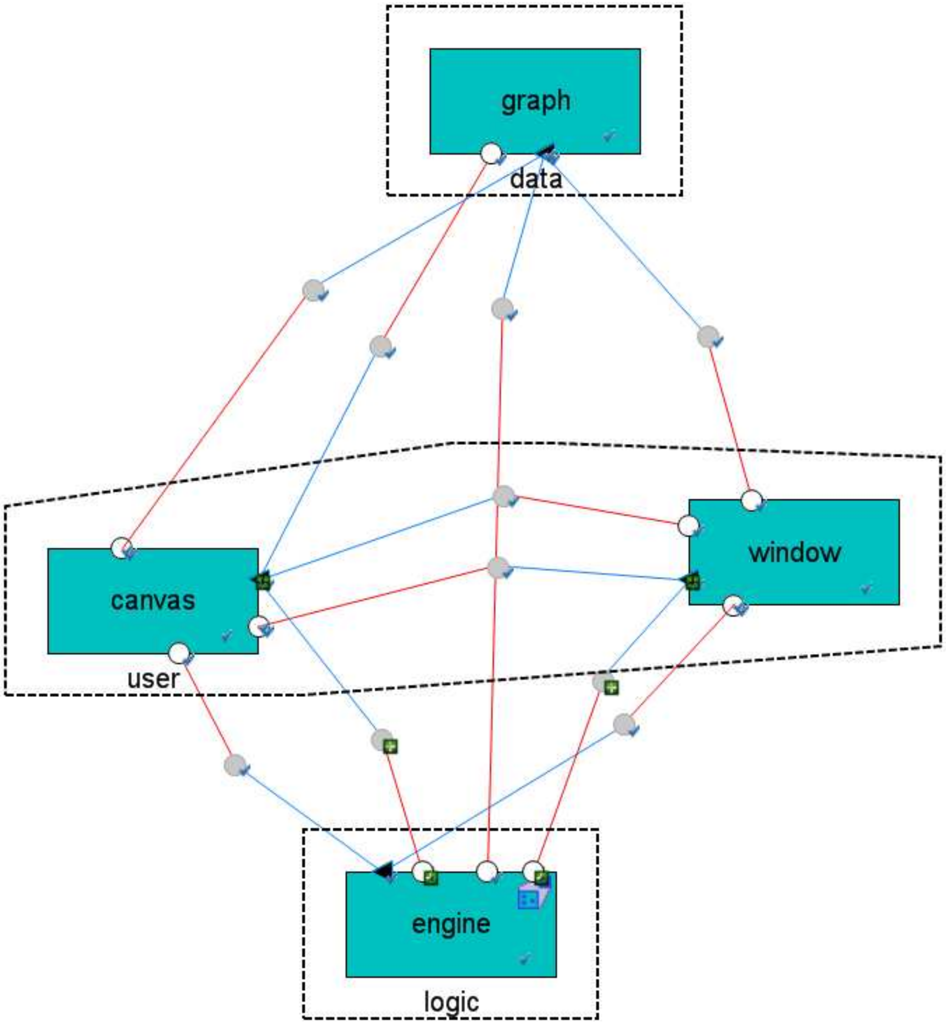



Figure 25: HillClimber conformance results. Component `engine` has two additional ports and is connected to components `window` and `canvas`.

# 8 Limitations and Future Work

**Structural vs. Full Conformance.** The approach is currently limited to checking the structural architectural conformance of an implementation to its design. In particular, this approach does not deal with architectural behavior.

**Architectural Dynamism.** The extracted architecture is an approximation of the actual execution architecture, one that is conservative and may include more than actually will be there, by virtue of using a sound static analysis. In particular, we do not currently represent structural dynamism, either in the as-designed architecture or in the as-built one.

However, the experimental evidence we have gathered on the two extended examples in this paper, as well as many other small examples, indicates that the extracted architectures do not suffer from too much or too little abstraction.

**Structural Matching Limitations.** The as-built and as-designed architecture must be structurally comparable. Comparing two very different views trivially deletes all elements of one view, and then inserts all elements from other view. To prevent this problem, detailed manual input to force or prevent matches may be needed. The algorithm does not currently detect split/merged nodes, or unrestricted moves. Finally, there are some limits on the scalability of the approach to large architecture. The algorithm is quadratic in the size of the trees, and currently suffers from excessive memory usage. But there is some room for optimization.

# 9 Related Work

**Architectural Recovery.** Few existing techniques recover the execution architecture. Most work at the level of the code architecture [54, 42, 64], which is easier to abstract.

Many architectural recovery approaches have been proposed, e.g., [32, 11, 31, 37]. Most approaches are supported by specialized workbenches: CIA [15], Rigi [52], Dali [39], Armin [56, 40], and many others [74, 66]. The main strength of these approaches is the ability to recover the architecture of systems made from many heterogeneous languages, and handle systems with an eroded architecture. These approaches typically recover an architecture using a mix of static and dynamic analysis, and require significant user input during the recovery process. Moreover, few approaches are incremental; i.e., changing the implementation often requires performing a completely recovery. In our approach, the annotations are added to the program to capture the architectural intent, and evolve with the program.

Architectural recovery approaches use various clustering mechanisms, e.g., [65, 46], or rely on naming conventions. In our approach, the grouping of objects is based on the architectural intent captured by the annotations, but also on the actual runtime execution structure.

**Design Enforcement.** Sangal et al. use design rules to enforce the code architecture, using package dependencies [64]. Our approach deals with the execution architecture.

Several approaches, e.g., SCL [34] and JavaCOP [8], enforce low-level, local programmer-oriented design intent. Our annotations and structural constraints are more architectural and global in nature.

**Reflexion Models.** Murphy et al. [54] also follow an incremental approach to check the as-built architecture against the as-designed one. The work on Reflexion Models however appears to be mostly concerned with module views, and not with C&C views. In Reflexion Models, the source model and the high-level models can be typed, partially typed or un-typed; similarly, assigning types is an optional step in our approach. We both support the same "goal of a lightweight technique by reducing the burden on the engineer to define a type for each high-level model interaction" with a "focus on those parts of the model where typing will provide the most benefit" - in our case, implementation-level violations of architectural intent. In Reflexion Models, a minimal representation of types is used, i.e., names, whereas Acme types have additional semantics and constraints associated with them. Just as Reflexion Models permit inconsistencies to remain, we allow the user to cancel any unwanted edit actions. Reflexion Models let the user elide information from view; we can also restrict the structural comparison to a subset of the underlying tree-structured data.

**Classification of architectural defects.** Roshandel et al. proposed a classification of architectural defects [63]. In this work, we focus on *topological errors*: "Topological errors tend to be global to the architecture and concern aspects related to the configuration of components and connectors in the system. They are often a result of the violation of constraints imposed by architectural styles. Some topological errors are directional in nature: the specific direction of communication required by the style is violated. [...] Other topological errors are structural in nature and are further divided into *usage* violations and *incompleteness* of the specification. [...] Incompleteness manifests itself when there is insufficient information for specifying the properties of the architectures components and connectors."

**Inconsistency Management.** There is significant work in the area of viewpoints, view merging and inconsistency management, e.g., [24]. A viewpoint captures data from disparate sources into independent but interrelated units. In view merging, there is also a notion of knowledge order or degree, i.e., a match can be disputed. When synchronizing between an as-built and an as-designed architecture, one may want to model incompleteness and inconsistency as a first class notion. In our approach, we model both views using the same viewtype, arbitrarily bridging the inevitable expressiveness gaps in the process. We also assume that one of the two views is authoritative. Implicitly, when the user decides to commit some edit actions but not others, they are allowing some acceptable differences to remain. In future work, it may be interesting to model this more precisely using ideas from inconsistency management.

**Object-Oriented Metrics.** There are several metrics for object-oriented design [16]. Many metrics deal with the code architecture, e.g., classes, and the number of methods. More recently, there has been a growing interest in measuring runtime coupling, using a dynamic analysis [9] or a static analysis [44].

To the best of our knowledge, this work is the first to look at a conformance metric relating the as-designed architecture to one obtained from the implementation.

**Architectural Metrics.** Researchers have proposed metrics for software architectural evolvability [73]. Such metrics are complementary to our approach and can guide the architectural types, styles and constraints that may be defined at the architectural level to limit architectural drift and erosion.

# 10 Conclusion

Previous attempts to relate the architecture to the implementation called for developing programs on ADL-specific implementation frameworks [49], or specifying the architecture directly in code, as in ArchJava [7]. Such proposals impose implementation restrictions or non-backward compatible language extensions. Most software developed today must be compatible with or use legacy systems, which often do not have documented architectures. We have a serious problem if we cannot determine the architecture of these systems for future software evolution. But at the same, re-engineering existing Java implementations to ArchJava would be prohibitively expensive for the millions of lines of existing code that power our information age. This report showed some initial results to address the problem for existing object-oriented languages and existing designs.

Today, most architectural recovery approaches use a mix of dynamic and static information, such as naming conventions, directory structures, etc. They often require the extractors to "play detective" [39], and involve some trial and error. Even so, existing compile-time approaches mostly obtain abstracted views of the *module* or *code architecture* [11, 33], but not the execution architecture.

In this report, we proposed a more principled approach. Developers add "simple" annotations to clarify the architectural intent in the code. These annotations are not radical language changes, and do not affect the program's runtime semantics. The annotations support existing languages, design idioms and patterns. The annotations do not require a specific implementation framework and can be used with existing frameworks and libraries. The annotations do not specify the architecture in code, like ArchJava did. Rather, they specify and enforce the sharing of data between objects, which has long been one of the challenges in extracting an execution architecture. Finally, the annotations enable the compile-time extraction of a sound execution architecture of a system from its annotated program.

Our structural differencing tool can check an implementation's structural conformance, by comparing the extracted as-built architecture to the as-designed architecture. We also developed a some initial ideas and measures of the structural conformance of an object-oriented implementation based on the structural comparison results.

To our knowledge, our approach is the first to statically assure a hierarchical execution architecture for object-oriented programs, written in existing languages, using existing libraries and general design idioms. In future work, we plan on obtaining more in-depth experience with the approach and the conformance measurements.

# Acknowledgements

This work was funded in part by the United States Department of Defense. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

The authors would like to acknowledge fruitful discussions with William Scherlis and Larry Maccherone. The authors also acknowledge Bradley Schmerl for his help with AcmeStudio.

# References


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

[2] M. Abi-Antoun and J. Aldrich. Ownership Domains in the Real World. In *International Workshop on Aliasing, Confinement and Ownership (IWACO)*, pages 93–104, 2007.

[3] 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):240–264, 2007.

[4] M. Abi-Antoun, J. Aldrich, N. Nahas, B. Schmerl, and D. Garlan. Differencing and Merging of Architectural Views. In *Automated Software Engineering*, pages 47–58, 2006.

[5] M. Abi-Antoun, J. Aldrich, N. Nahas, B. Schmerl, and D. Garlan. Differencing and Merging of Architectural Views. *Automated Software Engineering Journal*, 2007. Extended Version. To appear.

[6] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *European Conference on Object-Oriented Programming*, pages 1–25, 2004.

[7] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *International Conference on Software Engineering*, pages 187–197, 2002.

[8] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A Framework for Implementing Pluggable Type Systems. In *Object-Oriented Programming Systems, Languages, and Applications*, pages 57–74, 2006.

[9] E. Arisholm, L. C. Briand, and A. Foyen. Dynamic Coupling Measurement for Object-Oriented Software. *IEEE Trans. Softw. Eng.*, 30(8):491–506, 2004.

[10] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.

[11] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a Case Study: its Extracted Software Architecture. In *International Conference on Software Engineering*, pages 555–563, 1999.

[12] C. Boyapati, B. Liskov, and L. Shrira. Ownership Types for Object Encapsulation. In *Principles of Programming Languages*, pages 213–223, 2003.

[13] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: a System of Patterns*. John Wiley, 1996.

[14] P. H. Chen, M. Critchlow, A. Garg, C. van der Westhuizen, and A. van der Hoek. Differencing and Merging within an Evolving Product Line Architecture. In *Proceedings of the 5th International Workshop on Software Product-Family Engineering*, pages 269–281, 2003.

[15] Y.-F. Chen, M. Nishimoto, and C. Ramamoorthy. The C Information Abstraction System. *IEEE Trans. on Software Engineering*, 16(3):325–334, 1990.

[16] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object-Oriented Design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[17] D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *Object-Oriented Programming Systems, Languages, and Applications*, October 1998.

[18] P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architecture: View and Beyond*. Addison-Wesley, 2003.

[19] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty Years of Graph Matching in Pattern Recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(3):265–298, 2004.

[20] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. A Highly-Extensible, XML-Based Architecture Description Language. In *WICSA*, 2001.

[21] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the Execution of Java Programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, 2002.

[22] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *European Conference on Object-Oriented Programming*, pages XX–XX, 2007.

[23] W. Dietl and P. Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[24] S. Easterbrook and B. Nuseibeh. Using ViewPoints for Inconsistency Management. *Software Engineering Journal*, 11(1):31–43, 1996.

[25] C. Flanagan and S. N. Freund. Dynamic Architecture Extraction. In *Workshop on Formal Approaches to Testing and Runtime Verification*, August 2006.

[26] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Programming Language Design and Implementation*, pages 234–245, 2002.

[27] E. Gamma. Advanced Design with Patterns and Java (Tutorial). In *European Conference on Java and Object Orientation (JAOO)*, 1998.

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

[29] Gamma, E. et al. JHotDraw. `http://www.jhotdraw.org/`, 1996.

[30] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[31] G. Y. Guo, J. M. Atlee, and R. Kazman. A Software Architecture Reconstruction Method. In *Working IFIP Conference on Software Architecture (WICSA)*, pages 15–34, 1999.

[32] D. R. Harris, H. B. Reubenstein, and A. S. Yeh. Reverse engineering to the Architectural Level. In *International Conference on Software Engineering*, pages 186–195, 1995.

[33] A. E. Hassan and R. C. Holt. Architecture Recovery of Web Applications. In *International Conference on Software Engineering*, pages 349–359, 2002.

[34] H. J. Hoover and D. Hou. Using scl to specify and check design intent in source code. *IEEE Trans. Softw. Eng.*, 32(6):404–423, 2006.

[35] D. Jackson and M. Rinard. Software Analysis: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, 2000.

[36] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *IEEE Trans. on Software Engineering*, 27(2):156–169, 2001.

[37] D. Jerding and S. Rugaber. Using Visualization for Architectural Localization and Extraction. *Science of Computer Programming*, 36(2):267–284, 2000.

[38] W. Kaiser. Become a programming Picasso with JHotDraw. JavaWorld, February 2001.

[39] R. Kazman and S. J. Carriére. Playing Detective: Reconstructing Software Architecture from Available Evidence. *Automated Software Engg.*, 6(2):107–138, 1999.

[40] R. Kazman, L. O'Brien, and C. Verhoef. Architecture Reconstruction Guidelines, Third Edition. Technical Report CMU/SEI-2002-TR-034, Software Engineering Institute, 2002.

[41] D. Kirk, M. Roper, and M. Wood. Identifying and Addressing Problems in Object-Oriented Framework Reuse. *Empirical Software Engineering*, 2006.

[42] R. Kollman, P. Selonen, E. Stroulia, T. Systä, and A. Zundorf. A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In *Working Conference on Reverse Engineering*, pages 22–32, 2002.

[43] P. Lam and M. Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *European Conference on Object-Oriented Programming*, pages 275–302, 2003.

[44] Y. Liu and A. Milanova. Static Analysis for Dynamic Coupling Measures. In *Conference of the Center for Advanced Studies on Collaborative research*, page 10, 2006.

[45] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems. *IEEE Trans. on Software Engineering*, 31(3):256–272, 2005.

[46] O. Maqbool and H. Babri. Hierarchical Clustering for Software Architecture Recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, 2007.

[47] J. McGarry, D. Card, C. Jones, B. Layman, E. Clark, J. Dean, and F. Hall. *Practical Software Measurement: Objective Information for Decision Makers*. Addison-Wesley Professional, 2001.

[48] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 24–32, 1996.

[49] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. on Software Engineering*, 26(1), 2000.

[50] A. Mehra, J. Grundy, and J. Hosking. A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 204–213, 2005.

[51] R. Monroe. Capturing Software Architecture Design Expertise with Armani. Technical Report CMU-CS-98-163R, Carnegie Mellon University, January 2001.

[52] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A Reverse-Engineering Approach to Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, 1993.

[53] P. Müller and A. Rudich. Ownership Transfer in Universe Types. In *Object-Oriented Programming Systems, Languages, and Applications*, 2007.

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

[55] J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In *European Conference on Object-Oriented Programming*, 1998.

[56] L. O'Brien, C. Stoermer, and C. Verhoef. Software Architecture Reconstruction: Practice Needs and Current Approaches. Technical Report CMU/SEI-2002-TR-024, Software Engineering Institute, 2002.

[57] R. W. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie Mellon University, 2001.

[58] Omondo. EclipseUML. `http://www.omondo.com/`, 2006.

[59] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[60] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic Ownership for Generic Java. In *Object-Oriented Programming Systems, Languages, and Applications*, pages 397–412, 2006.

[61] D. Riehle. *Framework Design: a Role Modeling Approach*. PhD thesis, Federal Institute of Technology Zurich, 2000.

[62] R. Roshandel, N. Medvidovic, and L. Golubchik. A Bayesian Model for Predicting Reliability of Software Systems at the Architectural Level. In *International Conference on Quality of Software Architectures*, 2007.

[63] R. Roshandel, B. Schmerl, N. Medvidovic, D. Garlan, and D. Zhang. Understanding Tradeoffs among Different Architectural Modeling Approaches. In *Working IEEE/IFIP Conference on Software Architecture*, pages 47–56, 2004.

[64] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using Dependency Models to Manage Complex Software Architecture. In *Object-Oriented Programming Systems, Languages, and Applications*, 2005.

[65] K. Sartipi and K. Kontogiannis. A User-Assisted Approach to Component Clustering. *Journal of Software Maintenance*, 15(4):265–295, 2003.

[66] K. Sartipi and K. Kontogiannis. On Modeling Software Architecture Recovery as Graph Matching. In *Proceedings of the 19th IEEE International Conference on Software Maintenance*, pages 224–234, 2003.

[67] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering Architectures from Running Systems. *IEEE Trans. on Software Engineering*, 32(7):454–466, 2006.

[68] M. Shaw and D. Garlan. *Software Architectures: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[69] D. Soni, R. L. Nord, and C. Hofmeister. Software Architecture in Industrial Applications. In *International Conference on Software Engineering*, pages 196–207, 1995.

[70] A. Spiegel. *Automatic Distribution of Object-Oriented Programs*. PhD thesis, FU Berlin, 2002.

[71] B. Spitznagel and D. Garlan. Architecture-Based Performance Analysis. In *Conference on Software Engineering and Knowledge Engineering (SEKE'98)*, 1998.

[72] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration. *J. Systems & Software*, 44(3), 1999.

[73] N. Subramanian and L. Chung. Process-Oriented Metrics for Software Architecture Evolvability. In *International Workshop on Principles of Software Evolution*, pages 65–70, 2003.

[74] A. Telea, A. Maccari, and C. Riva. An Open Visualization Toolkit for Reverse Architecting. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 3–10, 2002.

[75] L. G. Williams and C. U. Smith. Performance evaluation of software architectures. In *Proceedings of the 1st International Workshop on Software and Performance (WOSP)*, pages 164–177, 1998.